# Table of Contents


- Overview
- New and updated articles
- SQL Server Guides
  - Always On Availability Groups Troubleshooting and Monitoring Guide
  - Index Architecture and Design
  - Memory Management Architecture
    - Reading pages
    - Writing pages
  - Pages and Extents Architecture
  - Post-migration Validation and Optimization
  - Performance
  - Query Processing Architecture
  - Thread and Task Architecture
  - Transaction Log Architecture and Management Guide
  - Transaction Locking and Row Versioning Guide
- Backup and restore
- Blob
- Collations
- Configuration Manager
- Cursors
- Data collection
- Data compression
- Data tier applications
- Database mail
- Databases
- Develop Clients
  - CLR integration
  - Extended stored procedures programming
  - Extended stored procedures reference

- Express Localdb error messages
- Express Localdb instance APIs
- Native client
- Server management objects (SMO)
- SQLXML
- WMI Provider (Windows Management Instrumentation)
  - WMI provider configuration
  - WMI provider configuration classes
  - WMI provider server events
Errors and events
Event classes
Extended events
Graphs
Hierarchical Data
Import and export
In-memory OLTP
Indexes
JSON
Linked servers
Maintenance plans
Manage
- Database Lifecycle Management
- Administer Multiple Servers Using Central Management Servers
Joins
Partitions
Policy-based management
PolyBase
Replication
Resource Governor
Scripting
Search
Security

Sequence numbers
Service Broker
Showplan Operators
Spatial
SQL Trace
Statistics
Stored procedures
Subqueries
Synonyms
System catalog views
System compatibility views
System dynamic management views
System functions
System information schema views
System stored procedures
System tables
System views
Tables
Track changes
Triggers
User-defined functions
Views
xml
Tutorials
- Getting Started with the Database Engine
  - Lesson 1: Connecting to the Database Engine
  - Lesson 2: Connecting from Another Computer
- SQL Server Backup and Restore to Azure Blob Storage Service
- Signing Stored Procedures with a Certificate
- Ownership Chains and Context Switching
- Using the Microsoft Azure Blob storage service with SQL Server 2016 databases
  - Lesson 1: Create a stored access policy and a shared access signature on an Azure container

Lesson 2: Create a SQL Server credential using a shared access signature
Lesson 3: Database backup to URL
Lesson 4: Restore database to virtual machine from URL
Lesson 5: Backup database using file-snapshot backup
Lesson 6: Generate activity and backup log using file-snapshot backup
Lesson 7: Restore a database to a point in time
Lesson 8. Restore as new database from log backup
Lesson 9: Manage backup sets and file-snapshot backups

# Guidance for using Microsoft SQL relational databases

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

This section contains the features and tasks associated with Microsoft SQL relational databases, database objects, data types, and the mechanisms used to work with or manage data. This information is common to most versions of the SQL Server Database Engine. Individual topics may indicate that some features are limited to some platforms. For information on a specific platform, see the following links:

- On-premises SQL Server and SQL Server Configuration
- SQL Server on Linux Documentation
- Azure SQL Database
- Azure SQL Data Warehouse
- Parallel Data Warehouse

# New and Recently Updated: Relational Databases docs

5/1/2018 • 15 min to read • Edit Online

Nearly every day Microsoft updates some of its existing articles on its Docs.Microsoft.com documentation website. This article displays excerpts from recently updated articles. Links to new articles might also be listed.

This article is generated by a program that is rerun periodically. Occasionally an excerpt can appear with imperfect formatting, or as markdown from the source article. Images are never displayed here.

Recent updates are reported for the following date range and subject:

- *Date range of updates:* **2018-02-03** -to- **2018-04-28**
- *Subject area:* **Relational Databases**.

## New Articles Created Recently

The following links jump to new articles that have been added recently.

1. Joins (SQL Server)
2. Subqueries (SQL Server)
3. Set up replication distribution database in Always On availability group
4. SQL Data Discovery and Classification
5. Transaction Locking and Row Versioning Guide
6. sys.dm_os_job_object (Azure SQL Database)
7. Filestream and FileTable system stored procedures (Transact-SQL)

## Updated Articles with Excerpts

This section displays the excerpts of updates gathered from articles that have recently experienced a large update.

The excerpts displayed here appear separated from their proper semantic context. Also, sometimes an excerpt is separated from important markdown syntax that surrounds it in the actual article. Therefore these excerpts are for general guidance only. The excerpts only enable you to know whether your interests warrant taking the time to click and visit the actual article.

For these and other reasons, do not copy code from these excerpts, and do not take as exact truth any text excerpt. Instead, visit the actual article.

#### Compact List of Articles Updated Recently

This compact list provides links to all the updated articles that are listed in the Excerpts section.

1. Use a Format File to Skip a Table Column (SQL Server)
2. JSON data in SQL Server
3. Query Processing Architecture Guide

4. Tutorial: Prepare SQL Server For Replication - Publisher, Distributor, Subscriber
5. Tutorial: Configure Replication between Two Fully Connected Servers (Transactional)
6. Tutorial: Configure Replication between a Server and Mobile Clients (Merge)
7. Query with Full-Text Search
8. Transparent Data Encryption with Bring Your Own Key support for Azure SQL Database and Data Warehouse
9. PowerShell and CLI: Enable Transparent Data Encryption using your own key from Azure Key Vault
10. About Change Data Capture (SQL Server)

## 1. Use a Format File to Skip a Table Column (SQL Server)

*Updated: 2018-04-13* (Next)

### Using OPENROWSET(BULK...)

To use an XML format file to skip a table column by using `OPENROWSET(BULK...)`, you have to provide an explicit list of columns in the select list and also in the target table, as follows:

```
INSERT ...<column_list> SELECT <column_list> FROM OPENROWSET(BULK...)
```

The following example uses the `OPENROWSET` bulk rowset provider and the `myTestSkipCol2.xml` format file. The example bulk imports the `myTestSkipCol2.dat` data file into the `myTestSkipCol` table. The statement contains an explicit list of columns in the select list and also in the target table, as required.

In SSMS, run the following code. Update the file system paths for the location of the sample files on your computer.

```
USE WideWorldImporters;
GO
INSERT INTO myTestSkipCol
  (Col1,Col3)
    SELECT Col1,Col3
      FROM  OPENROWSET(BULK  'C:\myTestSkipCol2.Dat',
      FORMATFILE='C:\myTestSkipCol2.Xml'
       ) as t1 ;
GO
```

---

## 2. JSON data in SQL Server

*Updated: 2018-04-13* (Previous | Next)

JSON documents may have sub-elements and hierarchical data that cannot be directly mapped into the standard relational columns. In this case, you can flatten JSON hierarchy by joining parent entity with sub-arrays.

In the following example, the second object in the array has sub-array representing person skills. Every sub-object can be parsed using additional `OPENJSON` function call:

```
DECLARE @json NVARCHAR(MAX)
SET @json =
N'[
       { "id" : 2,"info": { "name": "John", "surname": "Smith" }, "age": 25 },
       { "id" : 5,"info": { "name": "Jane", "surname": "Smith", "skills": ["SQL", "C#", "Azure"] }, "dob":
"2005-11-04T12:00:00" }
 ]'

SELECT *
FROM OPENJSON(@json)
  WITH (id int 'strict $.id',
        firstName nvarchar(50) '$.info.name', lastName nvarchar(50) '$.info.surname',
        age int, dateOfBirth datetime2 '$.dob',
    skills nvarchar(max) '$.skills' as json)
    outer apply openjson( a.skills )
                     with ( skill nvarchar(8) '$' ) as b
```

**skills** array is returned in the first `OPENJSON` as original JSON text fragment and passed to another `OPENJSON` function using `APPLY` operator. The second `OPENJSON` function will parse JSON array and return string values as single column rowset that will be joined with the result of the first `OPENJSON`. The result of this query is shown in the following table:

**Results**

| ID | FIRSTNAME | LASTNAME | AGE | DATEOFBIRTH | SKILL |
|---|---|---|---|---|---|
| 2 | John | Smith | 25 | | |
| 5 | Jane | Smith | | 2005-11-04T12:00:00 | SQL |
| 5 | Jane | Smith | | 2005-11-04T12:00:00 | C# |
| 5 | Jane | Smith | | 2005-11-04T12:00:00 | Azure |

`OUTER APPLY OPENJSON` will join first level entity with sub-array and return flatten resultset. Due to JOIN, the second row will be repeated for every skill.

---

### 3. Query Processing Architecture Guide

*Updated: 2018-04-13* (Previous | Next)

#### Logical Operator Precedence

When more than one logical operator is used in a statement, `NOT` is evaluated first, then `AND`, and finally `OR`. Arithmetic, and bitwise, operators are handled before logical operators. For more information, see [Operator Precedence].

In the following example, the color condition pertains to product model 21, and not to product model 20, because `AND` has precedence over `OR`.

```
SELECT ProductID, ProductModelID
FROM Production.Product
WHERE ProductModelID = 20 OR ProductModelID = 21
  AND Color = 'Red';
GO
```

You can change the meaning of the query by adding parentheses to force evaluation of the `OR` first. The following query finds only products under models 20 and 21 that are red.

```
SELECT ProductID, ProductModelID
FROM Production.Product
WHERE (ProductModelID = 20 OR ProductModelID = 21)
  AND Color = 'Red';
GO
```

Using parentheses, even when they are not required, can improve the readability of queries, and reduce the chance of making a subtle mistake because of operator precedence. There is no significant performance penalty in using parentheses. The following example is more readable than the original example, although they are syntactically the same.

```
SELECT ProductID, ProductModelID
FROM Production.Product
WHERE ProductModelID = 20 OR (ProductModelID = 21
  AND Color = 'Red');
GO
```

---

## 4. Tutorial: Prepare SQL Server For Replication - Publisher, Distributor, Subscriber

*Updated: 2018-04-13* (Previous | Next)

- Install SQL Server Management Studio.
- Install SQL Server 2017 Developer Edition.
- Download an AdventureWorks Sample Databases. For instructions on restoring a database in SSMS, please see Restoring a Database.

> **NOTE**
> - Replication is not supported on SQL Servers that are more than two versions apart. For more information, please see Supported SQL Versions in Repl Topology.
> - In *{Included-Content-Goes-Here}* , you must connect to the Publisher and Subscriber using a login that is a member of the **sysadmin** fixed server role. For more information on the sysadmin role, please see Server Level Roles.

**Estimated time to complete this tutorial: 30 minutes**

**Create Windows Accounts for Replication**

In this section, you will create Windows accounts to run replication agents. You will create a separate Windows account on the local server for the following agents:

| AGENT | LOCATION | ACCOUNT NAME |
| --- | --- | --- |
| Snapshot Agent | Publisher | <*machine_name*>\repl_snapshot |

## 5. Tutorial: Configure Replication between Two Fully Connected Servers (Transactional)

*Updated: 2018-04-13* (Previous | Next)

**Create a subscription to the Transactional publication**

In this section, you will add a subscriber to the Publication that was previously created. This tutorial uses a remote subscriber (NODE2\SQL2016) but a subscription can also be added locally to the publisher.

**To create the subscription**

1. Connect to the Publisher in *{Included-Content-Goes-Here}* , expand the server node, and then expand the **Replication** folder.
2. In the **Local Publications** folder, right-click the **AdvWorksProductTrans** publication, and then select **New Subscriptions**. The New Subscription Wizard launches:

   New Subscription

3. On the Publication page, select **AdvWorksProductTrans**, and then select **Next**:

   Select Tran Publisher

4. On the Distribution Agent Location page, select **Run all agents at the Distributor**, and then select **Next**. For more information on pull and push subscriptions, please see Subscribe to Publications:

   Run Agents at Dist

5. On the Subscribers page, if the name of the Subscriber instance is not displayed, select **Add Subscriber** and then select **Add SQL Server Subscriber** from the drop-down. This will launch the **Connect to Server** dialog box. Enter the Subscriber instance name and then select **Connect**.

## 6. Tutorial: Configure Replication between a Server and Mobile Clients (Merge)

*Updated: 2018-04-13* (Previous | Next)

The Employee table contains a column (OrganizationNode) that has the hierarchyid data type, which is only supported for replication in SQL 2017. If you're using a build lower than SQL 2017, you'll see a message at the bottom of the screen notifying you of potential data loss for using this column in bi-directional replication. For the purpose of this tutorial, this message can be ignored. However, this datatype should not be replicated in a production environment unless you're using the supported build. For more inforamtion about replicating the hierarchyid datatype, please see Using Hierarchyid Columns in Replication

- On the Filter Table Rows page, select **Add** and then select **Add Filter**.

- In the **Add Filter** dialog box, select **Employee (HumanResources)** in **Select the table to filter**. Select the **LoginID** column, select the right arrow to add the column to the WHERE clause of the filter query, and modify the WHERE clause as follows:

  ```
  WHERE [LoginID] = HOST_NAME()
  ```

  a. Select **A row from this table will go to only one subscription**, and select **OK**:

  Add Filter

- On the **Filter Table Rows** page, select **Employee (Human Resources)**, select **Add,** and then select **Add Join to Extend the Selected Filter**.

  a. In the **Add Join** dialog box, select **Sales.SalesOrderHeader** under **Joined table**. Select **Write the join statement manually**, and complete the join statement as follows:

---

## 7. Query with Full-Text Search

*Updated: 2018-04-13* (Previous | Next)

### More info about generation term searches

The *inflectional forms* are the different tenses and conjugations of a verb or the singular and plural forms of a noun.

For example, search for the inflectional form of the word "drive." If various rows in the table include the words "drive," "drives," "drove," "driving," and "driven," all would be in the result set because each of these can be inflectionally generated from the word drive.

[FREETEXT] and [FREETEXTTABLE] look for inflectional terms of all specified words by default. [CONTAINS] and [CONTAINSTABLE] support an optional `INFLECTIONAL` argument.

### Search for synonyms of a specific word

A *thesaurus* defines user-specified synonyms for terms. For more info about thesaurus files, see [Configure and Manage Thesaurus Files for Full-Text Search].

For example, if an entry, "{car, automobile, truck, van}," is added to a thesaurus, you can search for the thesaurus form of the word "car." All rows in the table queried that include the words "automobile," "truck," "van," or "car," appear in the result set because each of these words belongs to the synonym expansion set containing the word "car."

[FREETEXT] and [FREETEXTTABLE] use the thesaurus by default. [CONTAINS] and [CONTAINSTABLE] support an optional `THESAURUS` argument.

---

## 8. Transparent Data Encryption with Bring Your Own Key support for Azure SQL Database and Data Warehouse

*Updated: 2018-04-24* (Previous | Next)

**How to configure Geo-DR with Azure Key Vault**

To maintain high availability of TDE Protectors for encrypted databases, it is required to configure redundant Azure Key Vaults based on the existing or desired SQL Database failover groups or active geo-replication instances. Each geo-replicated server requires a separate key vault, that must be co-located with the server in the same Azure region. Should a primary database become inaccessible due to an outage in one region and a failover is triggered, the secondary database is able to take over using the secondary key vault.

For Geo-Replicated Azure SQL databases, the following Azure Key Vault configuration is required:

- One primary database with a key vault in region and one secondary database with a key vault in region.
- At least one secondary is required, up to four secondaries are supported.
- Secondaries of secondaries (chaining) are not supported.

The following section will go over the setup and configuration steps in more detail.

**Azure Key Vault Configuration Steps**

- Install PowerShell
- Create two Azure Key Vaults in two different regions using PowerShell to enable the "soft-delete" property on the key vaults (this option is not available from the AKV Portal yet – but required by SQL).
- Both Azure Key Vaults must be located in the two regions available in the same Azure Geo in order for backup and restore of keys to work. If you need the two key vaults to be located in different geos to meet SQL Geo-DR requirements, follow the BYOK Process that allows keys to be imported from an on-prem HSM.

---

## 9. PowerShell and CLI: Enable Transparent Data Encryption using your own key from Azure Key Vault

*Updated: 2018-04-24* (Previous | Next)

**Prerequisites for CLI**

- You must have an Azure subscription and be an administrator on that subscription.
- [Recommended but Optional] Have a hardware security module (HSM) or local key store for creating a local copy of the TDE Protector key material.
- Command-Line Interface version 2.0 or later. To install the latest version and connect to your Azure subscription, see Install and Configure the Azure Cross-Platform Command-Line Interface 2.0.
- Create an Azure Key Vault and Key to use for TDE.
  - Manage Key Vault using CLI 2.0
  - Instructions for using a hardware security module (HSM) and Key Vault
    - The key vault must have the following property to be used for TDE:
  - soft-delete
  - How to use Key Vault soft-delete with CLI
- The key must have the following attributes to be used for TDE:
  - No expiration date
  - Not disabled
  - Able to perform *get*, *wrap key*, *unwrap key* operations

**Step: Create a server and assign an Azure AD identity to your server**

```
cli
# create server (with identity) and database
```

## 10. About Change Data Capture (SQL Server)

*Updated: 2018-04-17* (Previous)

**Working with database and table collation differences**

It is important to be aware of a situation where you have different collations between the database and the columns of a table configured for change data capture. CDC uses interim storage to populate side tables. If a table has CHAR or VARCHAR columns with collations that are different from the database collation and if those columns store non-ASCII characters (such as double byte DBCS characters), CDC might not be able to persist the changed data consistent with the data in the base tables. This is due to the fact that the interim storage variables cannot have collations associated with them.

Please consider one of the following approaches to ensure change captured data is consistent with base tables:

- Use NCHAR or NVARCHAR data type for columns containing non-ASCII data.
- Or, Use the same collation for columns and for the database.

For example, if you have one database that uses a collation of SQL_Latin1_General_CP1_CI_AS, consider the following table:

```
CREATE TABLE T1(
     C1 INT PRIMARY KEY,
     C2 VARCHAR(10) collate Chinese_PRC_CI_AI)
```

CDC might fail to capture the binary data for column C2, because its collation is different (Chinese_PRC_CI_AI). Use NVARCHAR to avoid this problem:

```
CREATE TABLE T1(
     C1 INT PRIMARY KEY,
     C2 NVARCHAR(10) collate Chinese_PRC_CI_AI --Unicode data type, CDC works well with this data type)
```

# Similar articles about new or updated articles

This section lists very similar articles for recently updated articles in other subject areas, within our public GitHub.com repository: MicrosoftDocs/sql-docs.

**Subject areas that *do* have new or recently updated articles**

- New + Updated (11+6): **Advanced Analytics for SQL** docs
- New + Updated (18+0): **Analysis Services for SQL** docs
- New + Updated (218+14): **Connect to SQL** docs
- New + Updated (14+0): **Database Engine for SQL** docs
- New + Updated (3+2): **Integration Services for SQL** docs
- New + Updated (3+3): **Linux for SQL** docs
- New + Updated (7+10): **Relational Databases for SQL** docs

- New + Updated (0+2): **Reporting Services for SQL** docs
- New + Updated (1+3): **SQL Operations Studio** docs
- New + Updated (2+3): **Microsoft SQL Server** docs
- New + Updated (1+1): **SQL Server Data Tools (SSDT)** docs
- New + Updated (5+2): **SQL Server Management Studio (SSMS)** docs
- New + Updated (0+2): **Transact-SQL** docs
- New + Updated (1+1): **Tools for SQL** docs

**Subject areas that do *not* have any new or recently updated articles**

- New + Updated (0+0): **Analytics Platform System for SQL** docs
- New + Updated (0+0): **Data Quality Services for SQL** docs
- New + Updated (0+0): **Data Mining Extensions (DMX) for SQL** docs
- New + Updated (0+0): **Master Data Services (MDS) for SQL** docs
- New + Updated (0+0): **Multidimensional Expressions (MDX) for SQL** docs
- New + Updated (0+0): **ODBC (Open Database Connectivity) for SQL** docs
- New + Updated (0+0): **PowerShell for SQL** docs
- New + Updated (0+0): **Samples for SQL** docs
- New + Updated (0+0): **SQL Server Migration Assistant (SSMA)** docs
- New + Updated (0+0): **XQuery for SQL** docs

# SQL Server Guides

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

The following guides are available. They discuss general concepts and apply to all versions of SQL Server, unless stated otherwise in the respective guide.

Always On Availability Groups Troubleshooting and Monitoring Guide
Index Architecture and Design Guide
Memory Management Architecture Guide
Pages and Extents Architecture Guide
Post-migration Validation and Optimization Guide
Query Processing Architecture Guide
SQL Server Transaction Locking and Row Versioning Guide
SQL Server Transaction Log Architecture and Management Guide
Thread and Task Architecture Guide

# SQL Server Index Architecture and Design Guide

5/3/2018 • 60 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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. This SQL Server index design guide contains information on index architecture, and best practices to help you design effective indexes to meet the needs of your application.

This guide assumes the reader has a general understanding of the index types available in SQL Server. For a general description of index types, see Index Types.

This guide covers the following types of indexes:

- Clustered
- Nonclustered
- Unique
- Filtered
- Columnstore
- Hash
- Memory-Optimized Nonclustered

For information about XML indexes, see XML Indexes Overview.

For information about Spatial indexes, see Spatial Indexes Overview.

For information about Full-text indexes, see Populate Full-Text Indexes.

## 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* [1], or stored in a column-wise data format called *columnstore*.

The selection of the right indexes for a database and its workload is a complex balancing act between query speed and update cost. Narrow indexes, or indexes with few columns in the index key, require less disk space and maintenance overhead. Wide indexes, on the other hand, cover more queries. You may have to experiment with several different designs before finding the most efficient index. Indexes can be added, modified, and dropped without affecting the database schema or application design. Therefore, you should not hesitate to experiment with different indexes.

The query optimizer in SQL Server reliably chooses the most effective index in the vast majority of cases. Your overall index design strategy should provide a variety of indexes for the query optimizer to choose from and trust it to make the right decision. This reduces analysis time and produces good performance over a variety of situations. To see which indexes the query optimizer uses for a specific query, in SQL Server Management Studio, on the **Query** menu, select **Include Actual Execution Plan**.

Do not always equate index usage with good performance, and good performance with efficient index use. If using an index always helped produce the best performance, the job of the query optimizer would be simple. In reality, an incorrect index choice can cause less than optimal performance. Therefore, the task of the query optimizer is to select an index, or combination of indexes, only when it will improve performance, and to avoid indexed retrieval when it will hinder performance.

[1] Rowstore has been the traditional way to store relational table data. In SQL Server, rowstore refers to table where the underlying data storage format is a heap, a B-tree (clustered index), or a memory-optimized table.

**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 (12.x), memory-optimized tables and indexes are especially appropriate for this scenario, by providing a latch-free design. For more information, see Indexes for Memory-Optimized Tables, or Nonclustered Index for Memory-Optimized Tables Design Guidelines and Hash Index for Memory-Optimized Tables Design Guidelines in this guide.
   - 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 (11.x), 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. For more information, see Columnstore Indexes overview, or Columnstore Index Design Guidelines in this guide.
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. For more information, see Filtered Index Design Guidelines in this guide.
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. For more information, see Set Index Options.
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. For more information, see Partitioned Tables and Indexes. When you consider partitioning, determine whether the index should be aligned, that is, partitioned in essentially the same manner as the table, or partitioned independently.

# 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. For example, if a column is used in several indexes and you execute an `UPDATE` statement that modifies that column's data, each index that contains that column must be updated as well as the column in the underlying base table (heap or clustered index).
    - 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. For more information, see Database Engine Tuning Advisor.

**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. For example, a query of columns **a** and **b** on a table that has a composite index created on columns **a**, **b**, and **c** can retrieve the specified data from the index alone.
- 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. For

more information, see the section 'Index with Included Columns' in this guide.

- An **xml** data type can only be a key column only in an XML index. For more information, see XML Indexes (SQL Server). SQL Server 2012 SP1 introduces a new type of XML index known as a Selective 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. For more information, see Selective XML Indexes (SXI).

- 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. For more information, see Unique Index Design Guidelines in this guide.

- 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. For more information about data distribution, see Statistics.

- 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. For more information, see Indexes on 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. The partition function defines how each row is mapped to a set of partitions based on the values of certain columns, called partitioning columns. A partition scheme specifies the mapping of the partitions to a set of filegroups.

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.

  For more information, see Partitioned Tables and Indexes.

### 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. For example, the buyers in the Adventure Works Cycles purchasing department have to evaluate the quality of products they purchase from vendors. The buyers are most interested in finding products sent by these vendors with a high rejection rate. As shown in the following query, retrieving the data to meet this criteria requires the `RejectedQty` column in the `Purchasing.PurchaseOrderDetail` table to be sorted in descending order (large to small) and the `ProductID` column to be sorted in ascending order (small to large).

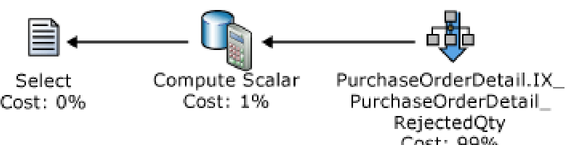
```
SELECT RejectedQty, ((RejectedQty/OrderQty)*100) AS RejectionRate,
    ProductID, DueDate
FROM Purchasing.PurchaseOrderDetail
ORDER BY RejectedQty DESC, ProductID ASC;
```

The following execution plan for this query shows that the query optimizer used a SORT operator to return the result set in the order specified by the ORDER BY clause.

If an index is created with key columns that match those in the ORDER BY clause in the query, the SORT operator can be eliminated in the query plan and the query plan is more efficient.

```
CREATE NONCLUSTERED INDEX IX_PurchaseOrderDetail_RejectedQty
ON Purchasing.PurchaseOrderDetail
    (RejectedQty DESC, ProductID ASC, DueDate, OrderQty);
```

After the query is executed again, the following execution plan shows that the SORT operator has been eliminated and the newly created nonclustered index is used.




The Database Engine can move equally efficiently in either direction. An index defined as `(RejectedQty DESC, ProductID ASC)` can still be used for a query in which the sort direction of the columns in the ORDER BY clause are reversed. For example, a query with the ORDER BY clause `ORDER BY RejectedQty ASC, ProductID DESC` can use the index.

Sort order can be specified only for key columns. The sys.index_columns catalog view and the INDEXKEY_PROPERTY function report whether an index column is stored in ascending or descending order.

# Metadata

Use these metadata views to see attributes of indexes. More architectural information is embedded in some of these views.

> **NOTE**
> For columnstore indexes, all columns are stored in the metadata as included columns. The columnstore index does not have key columns.

| | |
|---|---|
| sys.indexes (Transact-SQL) | sys.index_columns (Transact-SQL) |
| sys.partitions (Transact-SQL) | sys.internal_partitions (Transact-SQL) |
| sys.dm_db_index_operational_stats (Transact-SQL) | sys.dm_db_index_physical_stats (Transact-SQL) |

| | |
|---|---|
| sys.column_store_segments (Transact-SQL) | sys.column_store_dictionaries (Transact-SQL) |
| sys.column_store_row_groups (Transact-SQL) | sys.dm_db_column_store_row_group_operational_stats (Transact-SQL) |
| sys.dm_db_column_store_row_group_physical_stats (Transact-SQL) | sys.dm_column_store_object_pool (Transact-SQL) |
| sys.dm_db_column_store_row_group_operational_stats (Transact-SQL) | sys.dm_db_xtp_hash_index_stats (Transact-SQL) |
| sys.dm_db_xtp_index_stats (Transact-SQL) | sys.dm_db_xtp_object_stats (Transact-SQL) |
| sys.dm_db_xtp_nonclustered_index_stats (Transact-SQL) | sys.dm_db_xtp_table_memory_stats (Transact-SQL) |
| sys.hash_indexes (Transact-SQL) | sys.memory_optimized_tables_internal_attributes (Transact-SQL) |

## 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. With few exceptions, every table should have a clustered index defined on the column, or columns, that offer the following:

- 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

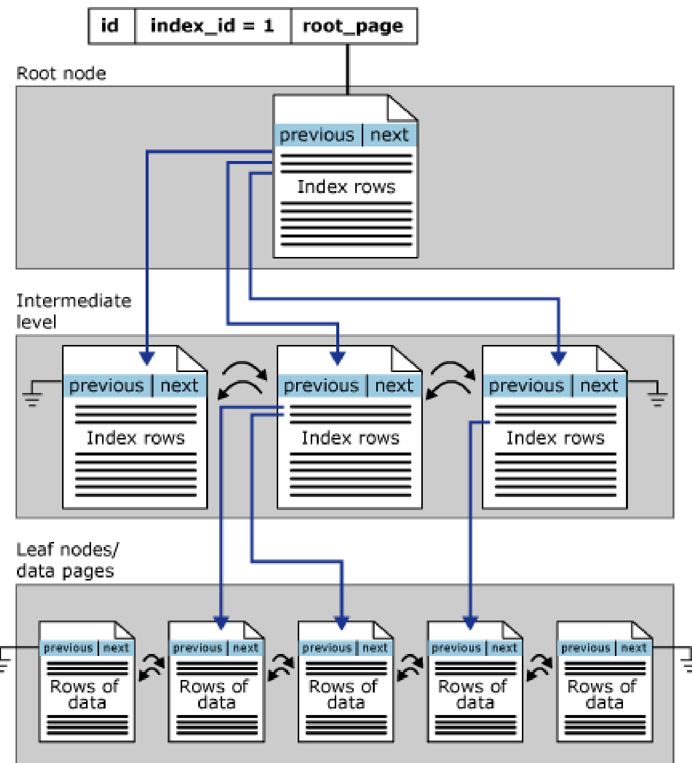
In SQL Server, indexes are organized as B-Trees. Each page in an index B-tree is called an index node. The top node of the B-tree is called the root node. The bottom nodes in the index are called the leaf nodes. Any index levels between the root and the leaf nodes are collectively known as intermediate 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. For example, if a clustered index has four partitions, there are four B-tree structures; one in each 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.

The pages in the data chain and the rows in them are ordered on the value of the clustered index key. All inserts are made at the point where the key value in the inserted row fits in the ordering sequence among existing rows.

This illustration shows the structure of a clustered index in a single partition.




### 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. Alternatively, a clustered index could be created on `LastName`, `FirstName`, `MiddleName` because employee records are frequently grouped and queried in this way, and the combination of these columns would still provide a high degree of difference.

  > **TIP**
  >
  > If not specified differently, when creating a PRIMARY KEY constraint, SQL Server creates a clustered index to support that constraint. Although a *uniqueidentifier* can be used to enforce uniqueness as a PRIMARY KEY, it is not an efficient clustering key. If using a *uniqueidentifier* as PRIMARY KEY, the recommendation is to create it as a nonclustered index, and use another column such as an `IDENTITY` to create the clustered index.

- Are accessed sequentially

  For example, a product ID uniquely identifies products in the `Production.Product` table in the **AdventureWorks2012** database. Queries in which a sequential search is specified, such as `WHERE ProductID BETWEEN 980 and 999`, would benefit from a clustered index on `ProductID`. This is because the rows would be stored in sorted order on that key column.

- 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. For example, to query the `HumanResources. Employee` table for all employees that report to a specific manager, the query optimizer might use the nonclustered index `IX_Employee_ManagerID`; this has `ManagerID` as its key column. The

query optimizer can quickly find all entries in the index that match the specified `ManagerID`. Each index entry points to the exact page and row in the table, or clustered index, in which the corresponding data can be found. After the query optimizer finds all entries in the index, it can go directly to the exact page and row to retrieve the data.

#### 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.

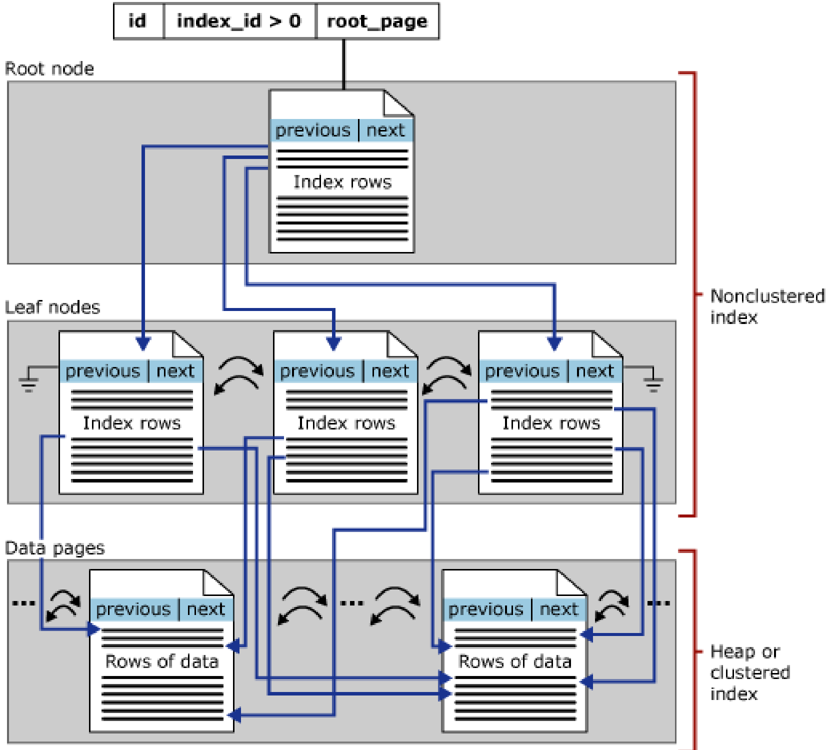
  The row locators in nonclustered index rows are either a pointer to a row or are a clustered index key for a row, as described in the following:

- 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.

  The following illustration shows the structure of a nonclustered index in a single partition.



**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**

Consider columns that have one or more of these attributes:

- 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.

##### Using Included Columns to Avoid Size Limits

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.
For example, assume that you want to index the following columns in the `Document` table:

- `Title nvarchar(50)`
- `Revision nchar(5)`
- `FileName nvarchar(400)`

  Because the **nchar** and **nvarchar** data types require 2 bytes for each character, an index that contains these three columns would exceed the 900 byte size limitation by 10 bytes (455 * 2). By using the `INCLUDE` clause of the `CREATE INDEX` statement, the index key could be defined as (`Title, Revision`) and `FileName` defined as a nonkey column. In this way, the index key size would be 110 bytes (55 * 2), and the index would still contain all the required columns. The following statement creates such an index.

```
CREATE INDEX IX_Document_Title
ON Production.Document (Title, Revision)
INCLUDE (FileName);
```

##### Index with Included Columns Guidelines

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. For more information, see Indexes on Computed 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.

###### Column Size Guidelines

- 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.

###### Column Modification Guidelines

When you modify a table column that has been defined as an included column, the following restrictions apply:

- 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.

> **NOTE**
>
> These column modification restrictions also apply to index key columns.

###### 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 included nonkey columns. In this way, you will have all columns needed to cover the query, but the index key itself is small and efficient.

For example, assume that you want to design an index to cover the following query.

```
SELECT AddressLine1, AddressLine2, City, StateProvinceID, PostalCode
FROM Person.Address
WHERE PostalCode BETWEEN N'98000' and N'99999';
```

To cover the query, each column must be defined in the index. Although you could define all columns as key columns, the key size would be 334 bytes. Because the only column actually used as search criteria is the `PostalCode` column, having a length of 30 bytes, a better index design would define `PostalCode` as the key column and include all other columns as nonkey columns.

The following statement creates an index with included columns to cover the query.

```
CREATE INDEX IX_Address_PostalCode
ON Person.Address (PostalCode)
INCLUDE (AddressLine1, AddressLine2, City, StateProvinceID);
```

###### 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. For example, if you want to make sure that the values in the `NationalIDNumber` column in the `HumanResources.Employee` table are unique, when the primary key is `EmployeeID`, create a UNIQUE constraint on the `NationalIDNumber` column. If the user tries to enter the same value in that column for more than one employee, an error message is displayed and the duplicate value is not entered.

With multicolumn unique indexes, the index guarantees that each combination of values in the index key is unique. For example, if a unique index is created on a combination of `LastName`, `FirstName`, and `MiddleName` columns, no two rows in the table could have the same combination of values for these columns.

Both clustered and nonclustered indexes can be unique. Provided that the data in the column is unique, you can create both a unique clustered index and multiple unique nonclustered indexes on the same table.

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. There are no significant differences between creating a UNIQUE constraint and creating a unique index 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. However, you should create a UNIQUE or PRIMARY KEY constraint on the column when data integrity is the objective. By doing this the objective of the index will be clear.

**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. Creating a unique index (preferably by creating a UNIQUE constraint) is recommended in this case.
- A unique nonclustered index can contain included nonkey columns. For more information, see Index with Included 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.

**Applies to**: SQL Server 2008 through SQL Server 2017.

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.

### Design Considerations

In order to design effective filtered indexes, it is important to understand what queries your application uses and how they relate to subsets of your data. Some examples of data that have well-defined subsets are columns with mostly NULL values, columns with heterogeneous categories of values and columns with distinct ranges of values. The following design considerations give a variety of scenarios for when a filtered index can provide advantages over full-table indexes.

> **TIP**
>
> The nonclustered columnstore index definition supports using a filtered condition. To minimize the performance impact of adding a columnstore index on an OLTP table, use a filtered condition to create a nonclustered columnstore index on only the cold data of your operational workload.

#### Filtered Indexes for subsets of data

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.

For example, the `AdventureWorks2012` database has a `Production.BillOfMaterials` table with 2679 rows. The `EndDate` column has only 199 rows that contain a non-NULL value and the other 2480 rows contain NULL. The following filtered index would cover queries that return the columns defined in the index and that select only rows with a non-NULL value for `EndDate`.

```
CREATE NONCLUSTERED INDEX FIBillOfMaterialsWithEndDate
    ON Production.BillOfMaterials (ComponentID, StartDate)
    WHERE EndDate IS NOT NULL ;
GO
```

The filtered index `FIBillOfMaterialsWithEndDate` is valid for the following query. You can display the query execution plan to determine if the query optimizer used the filtered index.

```
SELECT ProductAssemblyID, ComponentID, StartDate
FROM Production.BillOfMaterials
WHERE EndDate IS NOT NULL
    AND ComponentID = 5
    AND StartDate > '20080101' ;
```

For more information about how to create filtered indexes and how to define the filtered index predicate expression, see Create Filtered Indexes.

**Filtered Indexes for heterogeneous data**

When a table has heterogeneous data rows, you can create a filtered index for one or more categories of data.

For example, the products listed in the `Production.Product` table are each assigned to a `ProductSubcategoryID`, which are in turn associated with the product categories Bikes, Components, Clothing, or Accessories. These categories are heterogeneous because their column values in the `Production.Product` table are not closely correlated. For example, the columns `Color`, `ReorderPoint`, `ListPrice`, `Weight`, `Class`, and `Style` have unique characteristics for each product category. Suppose that there are frequent queries for accessories which have subcategories between 27 and 36 inclusive. You can improve the performance of queries for accessories by creating a filtered index on the accessories subcategories as shown in the following example.

```
CREATE NONCLUSTERED INDEX FIProductAccessories
    ON Production.Product (ProductSubcategoryID, ListPrice)
        Include (Name)
WHERE ProductSubcategoryID >= 27 AND ProductSubcategoryID <= 36;
```

The filtered index `FIProductAccessories` covers the following query because the query

results are contained in the index and the query plan does not include a base table lookup. For example, the query predicate expression `ProductSubcategoryID = 33` is a subset of the filtered index predicate `ProductSubcategoryID >= 27` and `ProductSubcategoryID <= 36`, the `ProductSubcategoryID` and `ListPrice` columns in the query predicate are both key columns in the index, and name is stored in the leaf level of the index as an included column.

```
SELECT Name, ProductSubcategoryID, ListPrice
FROM Production.Product
WHERE ProductSubcategoryID = 33 AND ListPrice > 25.00 ;
```

**Key Columns**

It is a best practice to include a small number of key or included columns in a filtered index definition, and to

incorporate only the columns that are necessary for the query optimizer to choose the filtered index for the query execution plan. The query optimizer can choose a filtered index for the query regardless of whether it does or does not cover the query. However, the query optimizer is more likely to choose a filtered index if it covers the query.

In some cases, a filtered index covers the query without including the columns in the filtered index expression as key or included columns in the filtered index definition. The following guidelines explain when a column in the filtered index expression should be a key or included column in the filtered index definition. The examples refer to the filtered index, `FIBillOfMaterialsWithEndDate` that was created previously.

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. For example, `FIBillOfMaterialsWithEndDate` covers the following query because the query predicate is equivalent to the filter expression, and `EndDate` is not returned with the query results. `FIBillOfMaterialsWithEndDate` does not need `EndDate` as a key or included column in the filtered index definition.

```
SELECT ComponentID, StartDate FROM Production.BillOfMaterials
WHERE EndDate IS NOT NULL;
```

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. For example, `FIBillOfMaterialsWithEndDate` is valid for the following query because it selects a subset of rows from the filtered index. However, it does not cover the following query because `EndDate` is used in the comparison `EndDate > '20040101'`, which is not equivalent to the filtered index expression. The query processor cannot execute this query without looking up the values of `EndDate`. Therefore, `EndDate` should be a key or included column in the filtered index definition.

```
SELECT ComponentID, StartDate FROM Production.BillOfMaterials
WHERE EndDate > '20040101';
```

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. For example, `FIBillOfMaterialsWithEndDate` does not cover the following query because it returns the `EndDate` column in the query results. Therefore, `EndDate` should be a key or included column in the filtered index definition.

```
SELECT ComponentID, StartDate, EndDate FROM Production.BillOfMaterials
WHERE EndDate IS NOT NULL;
```

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.

**Data Conversion Operators in the Filter Predicate**

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.

The following example creates a table with a variety of data types.

```
USE AdventureWorks2012;
GO
CREATE TABLE dbo.TestTable (a int, b varbinary(4));
```

In the following filtered index definition, column `b` is implicitly converted to an integer data type for the purpose of comparing it to the constant 1. This generates error message 10611 because the conversion occurs on the left hand side of the operator in the filtered predicate.

```
CREATE NONCLUSTERED INDEX TestTabIndex ON dbo.TestTable(a,b)
WHERE b = 1;
```

The solution is to convert the constant on the right hand side to be of the same type as column `b`, as seen in the following example:

```
CREATE INDEX TestTabIndex ON dbo.TestTable(a,b)
WHERE b = CONVERT(Varbinary(4), 1);
```

Moving the data conversion from the left side to the right side of a comparison operator might change the meaning of the conversion. In the above example, when the CONVERT operator was added to the right side, the comparison changed from an integer comparison to a **varbinary** comparison.

# Columnstore Index Design Guidelines

A *columnstore index* is a technology for storing, retrieving and managing data by using a columnar data format, called a columnstore. For more information, refer to Columnstore Indexes overview.

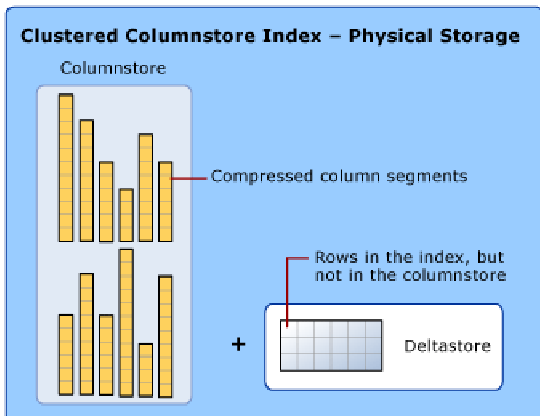
For version information, see Columnstore indexes - What's new.

### Columnstore Index Architecture

Knowing these basics will make it easier to understand other columnstore articles that explain how to use them effectively.

#### Data storage uses columnstore and rowstore compression

When discussing columnstore indexes, we use the terms *rowstore* and *columnstore* to emphasize the format for the data storage. Columnstore indexes use both types of storage.




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

A columnstore index physically stores most of the data in columnstore format. In columnstore format, the data is compressed and uncompressed as columns. There is no need to uncompress other values in each row that are not requested by the query. This makes it fast to scan an entire column of a large table.

- A **rowstore** is data that is logically organized as a table with rows and columns, and then physically stored in a row-wise data format. This has been the traditional way to store relational table data such as a heap or clustered B-tree index.

A columnstore index also physically stores some rows in a rowstore format called a deltastore. The deltastore,also called delta rowgroups, is a holding place for rows that are too few in number to qualify for compression into the columnstore. Each delta rowgroup is implemented as a clustered B-tree index.

- The **deltastore** is a holding place for rows that are too few in number to be compressed into the columnstore. The deltastore stores the rows in rowstore format.

#### Operations are performed on rowgroups and column segments

The columnstore index groups rows into manageable units. Each of these units is called a rowgroup. For best performance, the number of rows in a rowgroup is large enough to improve compression rates and small enough to benefit from in-memory operations.

- A **rowgroup** is a group of rows on which the columnstore index performs management and compression operations.

For example, the columnstore index performs these operations on rowgroups:

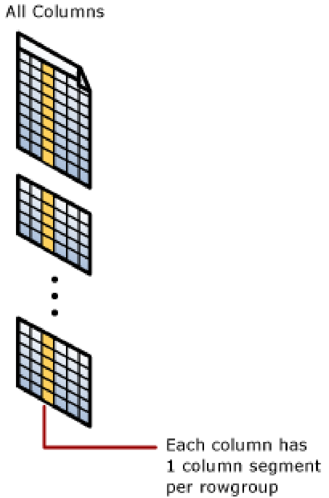
- Compresses rowgroups into the columnstore. Compression is performed on each column segment within a rowgroup.
- Merges rowgroups during an ALTER INDEX REORGANIZE operation.
- Creates new rowgroups during an ALTER INDEX REBUILD operation.
- Reports on rowgroup health and fragmentation in the dynamic management views (DMVs).

The deltastore is comprised of one or more rowgroups called delta rowgroups. Each delta rowgroup is a clustered B-tree index that stores rows when they are too few in number for compression into the columnstore.

- A **delta rowgroup** is a clustered B-tree index that stores small bulk loads and inserts until the rowgroup contains 1,048,576 rows or until the index is rebuilt. When a delta rowgroup contains 1,048,576 rows it is marked as closed and waits for a process called the tuple-mover to compress it into the columnstore.

Each column has some of its values in each rowgroup. These values are called column segments. When the columnstore index compresses a rowgroup, it compresses each column segment separately. To uncompress an entire column, the columnstore index only needs to uncompress one column segment from each rowgroup.

- A **column segment** is the portion of column values in a rowgroup. Each rowgroup contains one column segment for every column in the table. Each column has one column segment in each rowgroup.|




#### Small loads and inserts go to the deltastore

A columnstore index improves columnstore compression and performance by compressing at least 102,400 rows at a time into the columnstore index. To compress rows in bulk, the columnstore index accumulates small loads and inserts in the deltastore. The deltastore operations are handled behind the scenes. To return the correct query results, the clustered columnstore index combines query results from both the columnstore and the deltastore.

Rows go to the deltastore when they are:

- Inserted with the `INSERT INTO ... VALUES` statement.
- At the end of a bulk load and they number less than 102,400.
- Updated. Each update is implemented as a delete and an insert.

The deltastore also stores a list of IDs for deleted rows that have been marked as deleted but not yet physically deleted from the columnstore.

#### When delta rowgroups are full they get compressed into the columnstore

Clustered columnstore indexes collect up to 1,048,576 rows in each delta rowgroup before compressing the rowgroup into the columnstore. This improves the compression of the columnstore index. When a delta rowgroup contains 1,048,576 rows, the columnstore index marks the rowgroup as closed. A background process, called the *tuple-mover*, finds each closed rowgroup and compresses it into the columnstore.

You can force delta rowgroups into the columnstore by using ALTER INDEX to rebuild or reorganize the index. Note that if there is memory pressure during compression, the columnstore index might reduce the number of rows in the compressed rowgroup.

#### Each table partition has its own rowgroups and delta rowgroups

The concept of partitioning is the same in both a clustered index, a heap, and a columnstore index. Partitioning a table divides the table into smaller groups of rows according to a range of column values. It is often used for managing the data. For example, you could create a partition for each year of data, and then use partition switching to archive data to less expensive storage. Partition switching works on columnstore indexes and makes it easy to move a partition of data to another location.

Rowgroups are always defined within a table partition. When a columnstore index is partitioned, each partition has its own compressed rowgroups and delta rowgroups.

##### Each partition can have multiple delta rowgroups

Each partition can have more than one delta rowgroups. When the columnstore index needs to add data to a delta rowgroup and the delta rowgroup is locked, the columnstore index will try to obtain a lock on a different delta rowgroup. If there are no delta rowgroups available, the columnstore index will create a new delta rowgroup. For example, a table with 10 partitions could easily have 20 or more delta rowgroups.

#### You can combine columnstore and rowstore indexes on the same table

A nonclustered index contains a copy of part or all of the rows and columns in the underlying table. The index is defined as one or more columns of the table, and has an optional condition that filters the rows.

Starting with SQL Server 2016 (13.x), you can create an updatable **nonclustered columnstore index on a rowstore table**. The columnstore index stores a copy of the data so you do need extra storage. However, the data in the columnstore index will compress to a smaller size than the rowstore table requires. 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.

Starting with SQL Server 2016 (13.x), 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.

### Performance considerations

- The nonclustered columnstore index definition supports using a filtered condition. To minimize the performance impact of adding a columnstore index on an OLTP table, use a filtered condition to create a nonclustered columnstore index on only the cold data of your operational workload.
- An in-memory table can have one columnstore index. You can create it when the table is created or add it

later with ALTER TABLE (Transact-SQL). Before SQL Server 2016 (13.x), only a disk-based table could have a columnstore index.

For more information, refer to Columnstore indexes - Query performance.

**Design Guidance**

- A rowstore table can have one updateable nonclustered columnstore index. Before SQL Server 2014 (12.x), the nonclustered columnstore index was read-only.

For more information, refer to Columnstore indexes - Design Guidance.

# 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. For more information, see Indexes for Memory-Optimized Tables.

**Applies to**: SQL Server 2014 (12.x) through SQL Server 2017.

**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 memory-optimized 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.

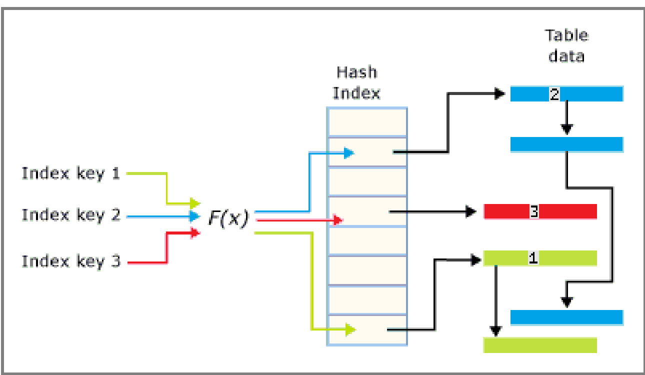
> **TIP**
> To determine the right `BUCKET_COUNT` for your data, see Configuring the hash index bucket count.

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 interplay of the hash index and the buckets is summarized in the following image.




### Configuring the hash index bucket count

The hash index bucket count is specified at index create time, and can be changed using the `ALTER TABLE...ALTER INDEX REBUILD` syntax.

In most cases the bucket count would ideally be between 1 and 2 times the number of distinct values in the index key.
You may not always be able to predict how many values a particular index key may have, or will have. Performance is usually still good if the **BUCKET_COUNT** value is within 10 times of the actual number of key values, and overestimating is generally better than underestimating.

Too **few** buckets has the following drawbacks:

- More hash collisions of distinct key values.
- Each distinct value is forced to share the same bucket with a different distinct value.
- The average chain length per bucket grows.
- The longer the bucket chain, the slower the speed of equality lookups in the index.

Too **many** buckets has the following drawbacks:

- Too high a bucket count might result in more empty buckets.
- Empty buckets impact the performance of full index scans. If those are performed regularly, consider picking a bucket count close to the number of distinct index key values.
- Empty buckets use memory, though each bucket uses only 8 bytes.

> **NOTE**
>
> Adding more buckets does nothing to reduce the chaining together of entries that share a duplicate value. The rate of value duplication is used to decide whether a hash is the appropriate index type, not to calculate the bucket count.

### Performance considerations

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.

**TIP**

The predicate must include all columns in the hash index key. The hash index requires a key (to hash) to seek into the index. If an index key consists of two columns and the `WHERE` clause only provides the first column, SQL Server does not have a complete key to hash. This will result in an index scan query plan.

If a hash index is used and the number of unique index keys is 100 times (or more) than the row count, consider either increasing to a larger bucket count to avoid large row chains, or use a nonclustered index instead.

**Declaration considerations**

A hash index can exist only on a memory-optimized table. It cannot exist on a disk-based table.

A hash index can be declared as:

- UNIQUE, or can default to Non-Unique.
- NONCLUSTERED, which is the default.

The following is an example of the syntax to create a hash index, outside of the CREATE TABLE statement:

````
```sql
ALTER TABLE MyTable_memop
ADD INDEX ix_hash_Column2 UNIQUE
HASH (Column2) WITH (BUCKET_COUNT = 64);
```
````

**Row versions and garbage collection**

In a memory-optimized table, when a row is affected by an `UPDATE`, the table creates an updated version of the row. During the update transaction, other sessions might be able to read the older version of the row and thereby avoid the performance slowdown associated with a row lock.

The hash index might also have different versions of its entries to accommodate the update.

Later when the older versions are no longer needed, a garbage collection (GC) thread traverses the buckets and their link lists to clean away old entries. The GC thread performs better if the link list chain lengths are short. For more information, refer to In-Memory OLTP Garbage Collection.

# Memory-Optimized Nonclustered Index Design Guidelines

Nonclustered indexes are one of the possible index types in a memory-optimized table. For more information, see Indexes for Memory-Optimized Tables.

**Applies to**: SQL Server 2014 (12.x) through SQL Server 2017.

**In-memory Nonclustered Index Architecture**

In-memory nonclustered indexes are implemented using a data structure called a Bw-Tree, originally envisioned and described by Microsoft Research in 2011. A Bw-Tree is a lock and latch-free variation of a B-Tree. For more details please see The Bw-Tree: A B-tree for New Hardware Platforms.

At a very high level the Bw-Tree can be understood as a map of pages organized by page ID (PidMap), a facility to allocate and reuse page IDs (PidAlloc) and a set of pages linked in the page map and to each other. These three high level sub-components make up the basic internal structure of a Bw-Tree.

The structure is similar to a normal B-Tree in the sense that each page has a set of key values that are ordered and there are levels in the index each pointing to a lower level and the leaf levels point to a data row. However there are several differences.

Just like hash indexes, multiple data rows can be linked together (versions). The page pointers between the levels are logical page IDs, which are offsets into a page mapping table, that in turn has the physical address for each page.

There are no in-place updates of index pages. New delta pages are introduced for this purpose.

- No latching or locking is required for page updates.
- Index pages are not a fixed size.

The key value in each non-leaf level page depicted is the highest value that the child that it points to contains and each row also contains that page logical page ID. On the leaf-level pages, along with the key value, it contains the physical address of the data row.

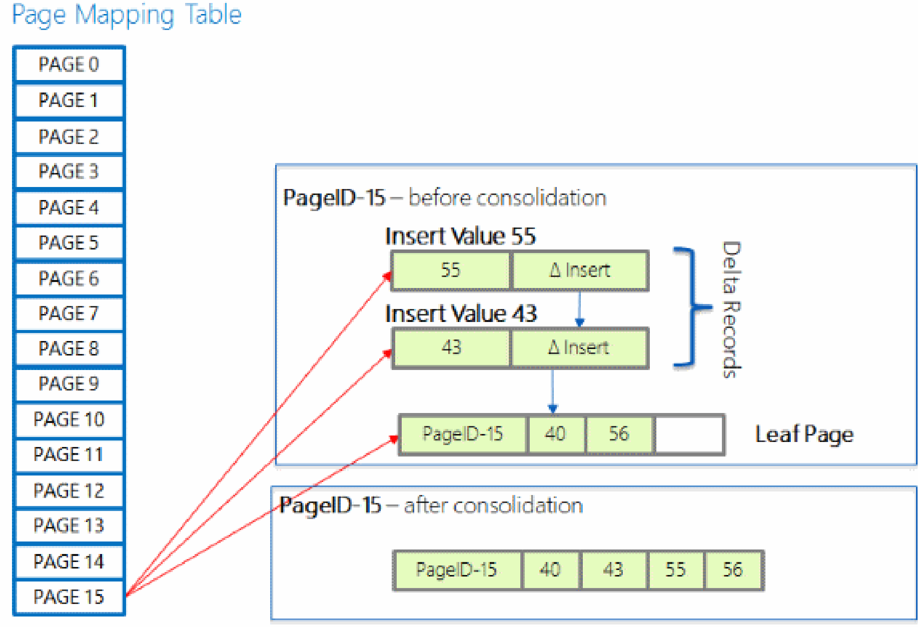
Point lookups are similar to B-Trees except that because pages are linked in only one direction, the SQL Server Database Engine follows right page pointers, where each non-leaf pages has the highest value of its child, rather than lowest value as in a B-Tree.

If a Leaf-level page has to change, the SQL Server Database Engine does not modify the page itself. Rather, the SQL Server Database Engine creates a delta record that describes the change, and appends it to the previous page. Then it also updates the page map table address for that previous page, to the address of the delta record which now becomes the physical address for this page.

There are three different operations that can be required for managing the structure of a Bw-Tree: consolidation, split and merge.

**Delta Consolidation**

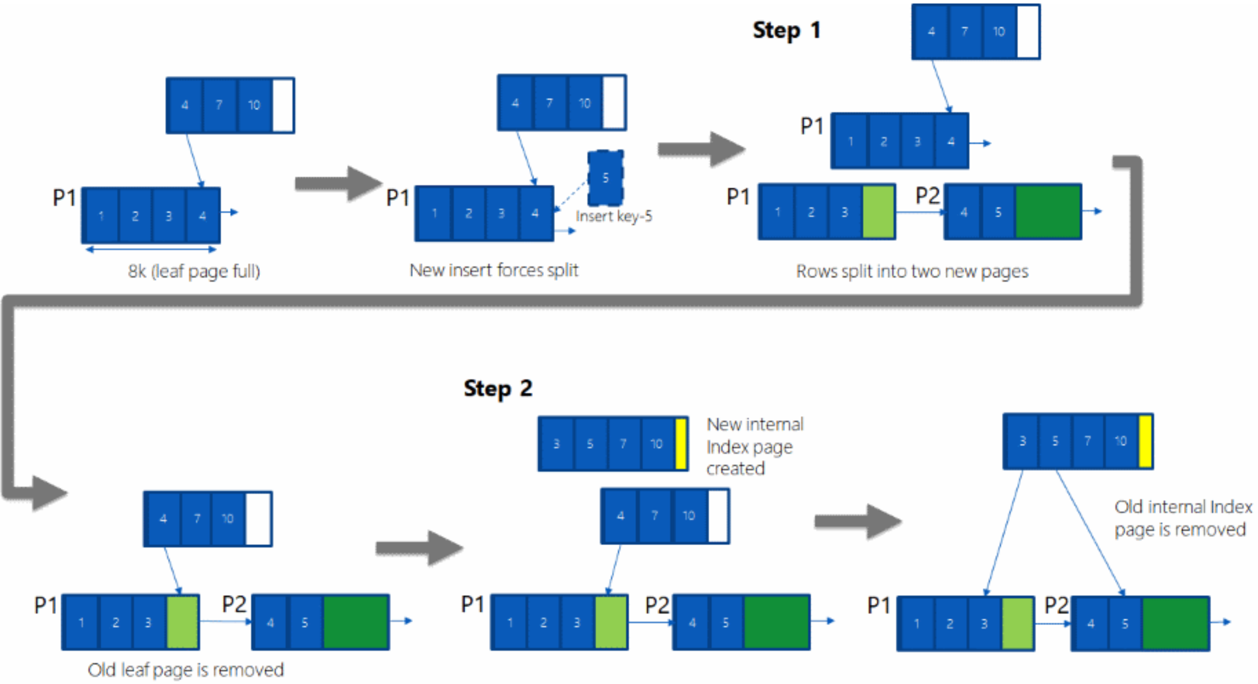
A long chain of delta records can eventually degrade search performance as it could mean we are traversing long chains when searching through an index. If a new delta record is added to a chain that already has 16 elements, the changes in the delta records will be consolidated into the referenced index page, and the page will then be rebuilt, including the changes indicated by the new delta record that triggered the consolidation. The newly rebuilt page will have the same page ID but a new memory address.




**Split page**

An index page in Bw-Tree grows on as-needed basis starting from storing a single row to storing a maximum of 8 KB. Once the index page grows to 8 KB, a new insert of a single row will cause the index page to split. For an internal page, this means when there is no more room to add another key value and pointer, and for a leaf page, it means that the row would be too big to fit on the page once all the delta records are incorporated. The statistics information in the page header for a leaf page keeps track of how much space would be required to consolidate the delta records, and that information is adjusted as each new delta record is added.

A Split operation is done in two atomic steps. In the picture below, assume a Leaf-page forces a split because a key with value 5 is being inserted, and a non-leaf page exists pointing to the end of the current Leaf-level page (key value 4).




**Step 1:** Allocate two new pages P1 and P2, and split the rows from old P1 page onto these new pages, including the newly inserted row. A new slot in Page Mapping Table is used to store the physical address of page P2. These pages, P1 and P2 are not accessible to any concurrent operations yet. In addition, the logical pointer from P1 to P2 is set. Then, in one atomic step update the Page Mapping Table to change the pointer from old P1 to new P1.

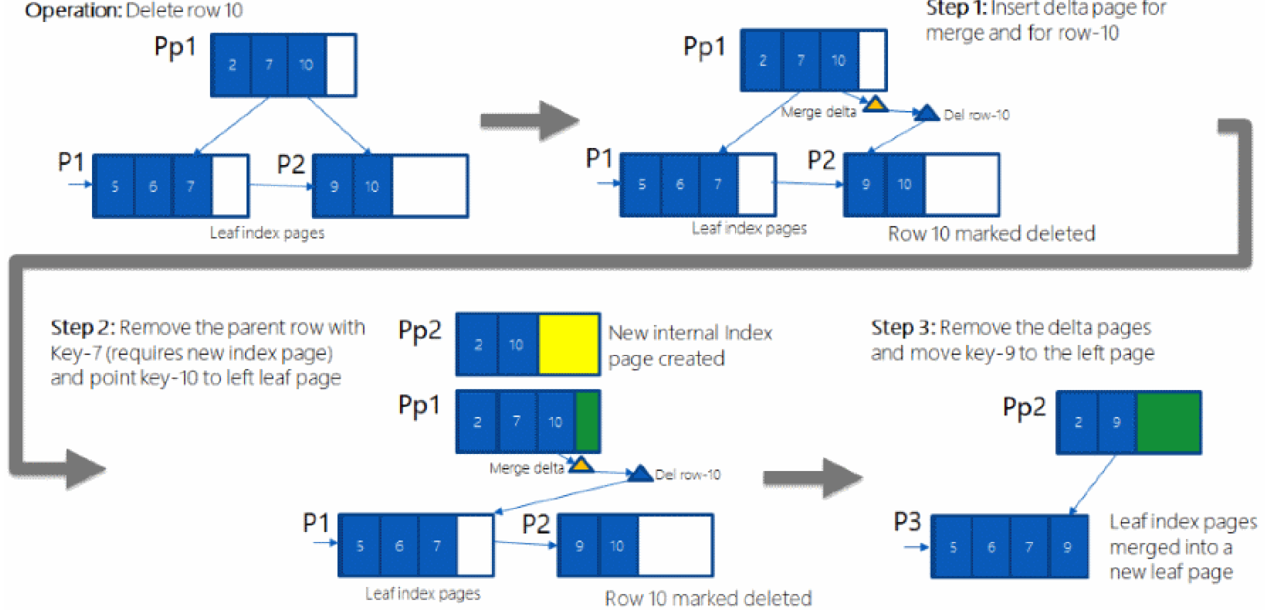
**Step 2:** The non-leaf page points to P1 but there is no direct pointer from a non-leaf page to P2. P2 is only reachable via P1. To create a pointer from a non-leaf page to P2, allocate a new non-leaf page (internal index page), copy all the rows from old non-leaf page, and add a new row to point to P2. Once this is done, in one atomic step, update the Page Mapping Table to change the pointer from old non-leaf page to new non-leaf page.

#### Merge page

When a `DELETE` operation results in a page having less than 10% of the maximum page size (currently 8 KB), or with a single row on it, that page will be merged with a contiguous page.

When a row is deleted from a page, a delta record for the delete is added. Additionally, a check is made to determine if the index page (non-leaf page) qualifies for Merge. This check verifies if the remaining space after deleting the row will be less than 10% of maximum page size. If it does qualify, the Merge is performed in three atomic steps.

In the picture below, assume a `DELETE` operation will delete the key value 10.




**Step 1:** A delta page representing key value 10 (blue triangle) is created and its pointer in the non-leaf page Pp1 is set to the new delta page. Additionally a special merge-delta page (green triangle) is created, and it is linked to point to the delta page. At this stage, both pages (delta page and merge-delta page) are not visible to any concurrent transaction. In one atomic step, the pointer to the Leaf-level page P1 in the Page Mapping Table is updated to point to the merge-delta page. After this step, the entry for key value 10 in Pp1 now points to the merge-delta page.

**Step 2:** The row representing key value 7 in the non-leaf page Pp1 needs to be removed, and the entry for key value 10 updated to point to P1. To do this, a new non-leaf page Pp2 is allocated and all the rows from Pp1 are copied except for the row representing key value 7; then the row for key value 10 is updated to point to page P1. Once this is done, in one atomic step, the Page Mapping Table entry pointing to Pp1 is updated to point to Pp2. Pp1 is no longer reachable.

**Step 3:** The Leaf-level pages P2 and P1 are merged and the delta pages removed. To do this, a new page P3 is allocated and the rows from P2 and P1 are merged, and the delta page changes are included in the new P3. Then, in one atomic step, the Page Mapping Table entry pointing to page P1 is updated to point to page P3.

### Performance considerations

The performance of a nonclustered index is better than nonclustered hash indexes when querying a memory-optimized table with inequality predicates.

> **NOTE**
>
> A column in a memory-optimized table can be part of both a hash index and a nonclustered index.

> **TIP**
>
> When a column in a nonclustered index key columns have many duplicate values, performance can degrade for updates, inserts, and deletes. One way to improve performance in this situation is to add another column to the nonclustered index.

## Additional Reading

Improving Performance with SQL Server 2008 Indexed Views
Partitioned Tables and Indexes
Create a Primary Key

Indexes for Memory-Optimized Tables
Columnstore Indexes overview
Troubleshooting Hash Indexes for Memory-Optimized Tables
Memory-Optimized Table Dynamic Management Views (Transact-SQL)
Index Related Dynamic Management Views and Functions (Transact-SQL)
Indexes on Computed Columns
Indexes and ALTER TABLE
CREATE INDEX (Transact-SQL)
ALTER INDEX (Transact-SQL)
CREATE XML INDEX (Transact-SQL)
CREATE SPATIAL INDEX (Transact-SQL)

# Memory Management Architecture Guide

5/3/2018 • 24 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

## Windows Virtual Memory Manager

The committed regions of address space are mapped to the available physical memory by the Windows Virtual Memory Manager (VMM).

For more information on the amount of physical memory supported by different operating systems, see the Windows documentation on Memory Limits for Windows Releases.

Virtual memory systems allow the over-commitment of physical memory, so that the ratio of virtual-to-physical memory can exceed 1:1. As a result, larger programs can run on computers with a variety of physical memory configurations. However, using significantly more virtual memory than the combined average working sets of all the processes can cause poor performance.

## SQL Server Memory Architecture

SQL Server dynamically acquires and frees memory as required. Typically, an administrator does not have to specify how much memory should be allocated to SQL Server, although the option still exists and is required in some environments.

One of the primary design goals of all database software is to minimize disk I/O because disk reads and writes are among the most resource-intensive operations. SQL Server builds a buffer pool in memory to hold pages read from the database. Much of the code in SQL Server is dedicated to minimizing the number of physical reads and writes between the disk and the buffer pool. SQL Server tries to reach a balance between two goals:

- Keep the buffer pool from becoming so big that the entire system is low on memory.
- Minimize physical I/O to the database files by maximizing the size of the buffer pool.

> **NOTE**
> In a heavily loaded system, some large queries that require a large amount of memory to run cannot get the minimum amount of requested memory and receive a time-out error while waiting for memory resources. To resolve this, increase the query wait Option. For a parallel query, consider reducing the max degree of parallelism Option.

> **NOTE**
> In a heavily loaded system under memory pressure, queries with merge join, sort and bitmap in the query plan can drop the bitmap when the queries do not get the minimum required memory for the bitmap. This can affect the query performance and if the sorting process can not fit in memory, it can increase the usage of worktables in tempdb database, causing tempdb to grow. To resolve this problem add physical memory or tune the queries to use a different and faster query plan.

#### Providing the maximum amount of memory to SQL Server

By using AWE and the Locked Pages in Memory privilege, you can provide the following amounts of memory to the SQL Server Database Engine.

> **NOTE**
> The following table includes a column for 32-bit versions, which are no longer available.

| | 32-BIT [1] | 64-BIT |
|---|---|---|
| Conventional memory | All SQL Server editions. Up to process virtual address space limit:<br>- 2 GB<br>- 3 GB with /3gb boot parameter [2]<br>- 4 GB on WOW64 [3] | All SQL Server editions. Up to process virtual address space limit:<br>- 7 TB with IA64 architecture (IA64 not supported in SQL Server 2012 (11.x) and above)<br>- Operating system maximum with x64 architecture [4] |
| AWE mechanism (Allows SQL Server to go beyond the process virtual address space limit on 32-bit platform.) | SQL Server Standard, Enterprise, and Developer editions: Buffer pool is capable of accessing up to 64 GB of memory. | Not applicable [5] |
| Lock pages in memory operating system (OS) privilege (allows locking physical memory, preventing OS paging of the locked memory.) [6] | SQL Server Standard, Enterprise, and Developer editions: Required for SQL Server process to use AWE mechanism. Memory allocated through AWE mechanism cannot be paged out. Granting this privilege without enabling AWE has no effect on the server. | Only used when necessary, namely if there are signs that sqlservr process is being paged out. In this case, error 17890 will be reported in the Errorlog, resembling the following example:<br>`A significant part of sql server process memory has been paged out. This may result in a performance degradation. Duration: #### seconds. Working set (KB): ####, committed (KB): ####, memory utilization: ##%.` |

[1] 32-bit versions are not available starting with SQL Server 2014 (12.x).
[2] /3gb is an operating system boot parameter. For more information, visit the MSDN Library.
[3] WOW64 (Windows on Windows 64) is a mode in which 32-bit SQL Server runs on a 64-bit operating system.
[4] SQL Server Standard Edition supports up to 128 GB. SQL Server Enterprise Edition supports the operating system maximum.
[5] Note that the sp_configure awe enabled option was present on 64-bit SQL Server, but it is ignored.
[6] If lock pages in memory privilege (LPIM) is granted (either on 32-bit for AWE support or on 64-bit by itself), we recommend also setting max server memory. For more information on LPIM, refer to Server Memory Server Configuration Options

> **NOTE**
> Older versions of SQL Server could run on a 32-bit operating system. Accessing more than 4 gigabytes (GB) of memory on a 32-bit operating system required Address Windowing Extensions (AWE) to manage the memory. This is not necessary when SQL Server is running on 64-bit operation systems. For more information about AWE, see Process Address Space and Managing Memory for Large Databases in the SQL Server 2008 documentation.

## Changes to Memory Management starting with SQL Server 2012 (11.x)

In earlier versions of SQL Server ( SQL Server 2005, SQL Server 2008 and SQL Server 2008 R2), memory allocation was done using five different mechanisms:

- **Single-page Allocator (SPA)**, including only memory allocations that were less than, or equal to 8-KB in the SQL Server process. The *max server memory (MB)* and *min server memory (MB)* configuration options determined the limits of physical memory that the SPA consumed. THe buffer pool was simultaneously the

mechanism for SPA, and the largest consumer of single-page allocations.

- **Multi-Page Allocator (MPA)**, for memory allocations that request more than 8-KB.
- **CLR Allocator**, including the SQL CLR heaps and its global allocations that are created during CLR initialization.
- Memory allocations for **thread stacks** in the SQL Server process.
- **Direct Windows allocations (DWA)**, for memory allocation requests made directly to Windows. These include Windows heap usage and direct virtual allocations made by modules that are loaded into the SQL Server process. Examples of such memory allocation requests include allocations from extended stored procedure DLLs, objects that are created by using Automation procedures (sp_OA calls), and allocations from linked server providers.

Starting with SQL Server 2012 (11.x), Single-lage allocations, Multi-Page allocations and CLR allocations are all consolidated into a **"Any size" Page Allocator**, and it's included in memory limits that are controlled by *max server memory (MB)* and *min server memory (MB)* configuration options. This change provided a more accurate sizing ability for all memory requirements that go through the SQL Server memory manager.

> **IMPORTANT**
> Carefully review your current *max server memory (MB)* and *min server memory (MB)* configurations after you upgrade to SQL Server 2012 (11.x) through SQL Server 2017. This is because starting in SQL Server 2012 (11.x), such configurations now include and account for more memory allocations compared to earlier versions. These changes apply to both 32-bit and 64-bit versions of SQL Server 2012 (11.x) and SQL Server 2014 (12.x), and 64-bit versions of SQL Server 2016 (13.x) through SQL Server 2017.

The following table indicates whether a specific type of memory allocation is controlled by the *max server memory (MB)* and *min server memory (MB)* configuration options:

| TYPE OF MEMORY ALLOCATION | SQL SERVER 2005, SQL SERVER 2008 AND SQL SERVER 2008 R2 | STARTING WITH SQL SERVER 2012 (11.X) |
| --- | --- | --- |
| Single-page allocations | Yes | Yes, consolidated into "any size" page allocations |
| Multi-page allocations | No | Yes, consolidated into "any size" page allocations |
| CLR allocations | No | Yes |
| Thread stacks memory | No | No |
| Direct allocations from Windows | No | No |

Starting with SQL Server 2012 (11.x), SQL Server might allocate more memory than the value specified in the max server memory setting. This behavior may occur when the ***Total Server Memory (KB)*** value has already reached the ***Target Server Memory (KB)*** setting (as specified by max server memory). If there is insufficient contiguous free memory to meet the demand of multi-page memory requests (more than 8 KB) because of memory fragmentation, SQL Server can perform over-commitment instead of rejecting the memory request.

As soon as this allocation is performed, the *Resource Monitor* background task starts to signal all memory consumers to release the allocated memory, and tries to bring the *Total Server Memory (KB)* value below the *Target Server Memory (KB)* specification. Therefore, SQL Server memory usage could briefly exceed the max server memory setting. In this situation, the *Total Server Memory (KB)* performance counter reading will exceed the max server memory and *Target Server Memory (KB)* settings.

This behavior is typically observed during the following operations:

- Large Columnstore index queries.
- Columnstore index (re)builds, which use large volumes of memory to perform Hash and Sort operations.
- Backup operations that require large memory buffers.
- Tracing operations that have to store large input parameters.

## Changes to "memory_to_reserve" starting with SQL Server 2012 (11.x)

In earlier versions of SQL Server ( SQL Server 2005, SQL Server 2008 and SQL Server 2008 R2), the SQL Server memory manager set aside a part of the process virtual address space (VAS) for use by the **Multi-Page Allocator (MPA)**, **CLR Allocator**, memory allocations for **thread stacks** in the SQL Server process, and **Direct Windows allocations (DWA)**. This part of the virtual address space is also known as "Mem-To-Leave" or "non-Buffer Pool" region.

The virtual address space that is reserved for these allocations is determined by the ***memory_to_reserve*** configuration option. The default value that SQL Server uses is 256 MB. To override the default value, use the SQL Server *-g* startup parameter. Refer to the documentation page on Database Engine Service Startup Options for information on the *-g* startup parameter.

Because starting with SQL Server 2012 (11.x), the new "any size" page allocator also handles allocations greater than 8 KB, the *memory_to_reserve* value does not include the multi-page allocations. Except for this change, everything else remains the same with this configuration option.

The following table indicates whether a specific type of memory allocation falls into the *memory_to_reserve* region of the virtual address space for the SQL Server process:

| TYPE OF MEMORY ALLOCATION | SQL SERVER 2005, SQL SERVER 2008 AND SQL SERVER 2008 R2 | STARTING WITH SQL SERVER 2012 (11.X) |
| --- | --- | --- |
| Single-page allocations | No | No, consolidated into "any size" page allocations |
| Multi-page allocations | Yes | No, consolidated into "any size" page allocations |
| CLR allocations | Yes | Yes |
| Thread stacks memory | Yes | Yes |
| Direct allocations from Windows | Yes | Yes |

## Dynamic Memory Management

The default memory management behavior of the SQL Server SQL Server Database Engine is to acquire as much memory as it needs without creating a memory shortage on the system. The SQL Server Database Engine does this by using the Memory Notification APIs in Microsoft Windows.

When SQL Server is using memory dynamically, it queries the system periodically to determine the amount of free memory. Maintaining this free memory prevents the operating system (OS) from paging. If less memory is free, SQL Server releases memory to the OS. If more memory is free, SQL Server may allocate more memory. SQL Server adds memory only when its workload requires more memory; a server at rest does not increase the size of its virtual address space.

**Max server memory** controls the SQL Server memory allocation, compile memory, all caches (including the

buffer pool), query execution memory grants, lock manager memory, and CLR[1] memory (essentially any memory clerk found in **sys.dm_os_memory_clerks**).

[1] CLR memory is managed under max_server_memory allocations starting with SQL Server 2012 (11.x).

The following query returns information about currently allocated memory:

```
SELECT
  physical_memory_in_use_kb/1024 AS sql_physical_memory_in_use_MB,
    large_page_allocations_kb/1024 AS sql_large_page_allocations_MB,
    locked_page_allocations_kb/1024 AS sql_locked_page_allocations_MB,
    virtual_address_space_reserved_kb/1024 AS sql_VAS_reserved_MB,
    virtual_address_space_committed_kb/1024 AS sql_VAS_committed_MB,
    virtual_address_space_available_kb/1024 AS sql_VAS_available_MB,
    page_fault_count AS sql_page_fault_count,
    memory_utilization_percentage AS sql_memory_utilization_percentage,
    process_physical_memory_low AS sql_process_physical_memory_low,
    process_virtual_memory_low AS sql_process_virtual_memory_low
FROM sys.dm_os_process_memory;
```

Memory for thread stacks[1], CLR[2], extended procedure .dll files, the OLE DB providers referenced by distributed queries, automation objects referenced in Transact-SQL statements, and any memory allocated by a non SQL Server DLL are **not** controlled by max server memory.

[1] Refer to the documentation page on how to Configure the max worker threads Server Configuration Option, for information on the calculated default worker threads for a given number of affinitized CPUs in the current host. SQL Server stack sizes are as follows:

| SQL SERVER ARCHITECTURE | OS ARCHITECTURE | STACK SIZE |
| --- | --- | --- |
| x86 (32-bit) | x86 (32-bit) | 512 KB |
| x86 (32-bit) | x64 (64-bit) | 768 KB |
| x64 (64-bit) | x64 (64-bit) | 2048 KB |
| IA64 (Itanium) | IA64 (Itanium) | 4096 KB |

[2] CLR memory is managed under max_server_memory allocations starting with SQL Server 2012 (11.x).

SQL Server uses the memory notification API **QueryMemoryResourceNotification** to determine when the SQL Server Memory Manager may allocate memory and release memory.

When SQL Server starts, it computes the size of virtual address space for the buffer pool based on a number of parameters such as amount of physical memory on the system, number of server threads and various startup parameters. SQL Server reserves the computed amount of its process virtual address space for the buffer pool, but it acquires (commits) only the required amount of physical memory for the current load.

The instance then continues to acquire memory as needed to support the workload. As more users connect and run queries, SQL Server acquires the additional physical memory on demand. A SQL Server instance continues to acquire physical memory until it either reaches its max server memory allocation target or Windows indicates there is no longer an excess of free memory; it frees memory when it has more than the min server memory setting, and Windows indicates that there is a shortage of free memory.

As other applications are started on a computer running an instance of SQL Server, they consume memory and the amount of free physical memory drops below the SQL Server target. The instance of SQL Server adjusts its memory consumption. If another application is stopped and more memory becomes available, the instance of SQL Server increases the size of its memory allocation. SQL Server can free and acquire several megabytes of

memory each second, allowing it to quickly adjust to memory allocation changes.

## Effects of min and max server memory

The min server memory and max server memory configuration options establish upper and lower limits to the amount of memory used by the buffer pool and other caches of the SQL Server Database Engine. The buffer pool does not immediately acquire the amount of memory specified in min server memory. The buffer pool starts with only the memory required to initialize. As the SQL Server Database Engine workload increases, it keeps acquiring the memory required to support the workload. The buffer pool does not free any of the acquired memory until it reaches the amount specified in min server memory. Once min server memory is reached, the buffer pool then uses the standard algorithm to acquire and free memory as needed. The only difference is that the buffer pool never drops its memory allocation below the level specified in min server memory, and never acquires more memory than the level specified in max server memory.

> **NOTE**
>
> SQL Server as a process acquires more memory than specified by max server memory option. Both internal and external components can allocate memory outside of the buffer pool, which consumes additional memory, but the memory allocated to the buffer pool usually still represents the largest portion of memory consumed by SQL Server.

The amount of memory acquired by the SQL Server Database Engine is entirely dependent on the workload placed on the instance. A SQL Server instance that is not processing many requests may never reach min server memory.

If the same value is specified for both min server memory and max server memory, then once the memory allocated to the SQL Server Database Engine reaches that value, the SQL Server Database Engine stops dynamically freeing and acquiring memory for the buffer pool.

If an instance of SQL Server is running on a computer where other applications are frequently stopped or started, the allocation and deallocation of memory by the instance of SQL Server may slow the startup times of other applications. Also, if SQL Server is one of several server applications running on a single computer, the system administrators may need to control the amount of memory allocated to SQL Server. In these cases, you can use the min server memory and max server memory options to control how much memory SQL Server can use. The **min server memory** and **max server memory** options are specified in megabytes. For more information, see Server Memory Configuration Options.

## Memory used by SQL Server objects specifications

The following list describes the approximate amount of memory used by different objects in SQL Server. The amounts listed are estimates and can vary depending on the environment and how objects are created:

- Lock (as maintained by the Lock Manager): 64 bytes + 32 bytes per owner
- User connection: Approximately (3 * network_packet_size + 94 kb)

The **network packet size** is the size of the tabular data scheme (TDS) packets that are used to communicate between applications and the SQL Server Database Engine. The default packet size is 4 KB, and is controlled by the network packet size configuration option.

When multiple active result sets (MARS) are enabled, the user connection is approximately (3 + 3 * num_logical_connections) * network_packet_size + 94 KB

## Buffer management

The primary purpose of a SQL Server database is to store and retrieve data, so intensive disk I/O is a core characteristic of the Database Engine. And because disk I/O operations can consume many resources and take a

relatively long time to finish, SQL Server focuses on making I/O highly efficient. Buffer management is a key component in achieving this efficiency. The buffer management component consists of two mechanisms: the **buffer manager** to access and update database pages, and the **buffer cache** (also called the **buffer pool**), to reduce database file I/O.

### How buffer management works

A buffer is an 8 KB page in memory, the same size as a data or index page. Thus, the buffer cache is divided into 8 KB pages. The buffer manager manages the functions for reading data or index pages from the database disk files into the buffer cache and writing modified pages back to disk. A page remains in the buffer cache until the buffer manager needs the buffer area to read in more data. Data is written back to disk only if it is modified. Data in the buffer cache can be modified multiple times before being written back to disk. For more information, see Reading Pages and Writing Pages.

When SQL Server starts, it computes the size of virtual address space for the buffer cache based on a number of parameters such as the amount of physical memory on the system, the configured number of maximum server threads, and various startup parameters. SQL Server reserves this computed amount of its process virtual address space (called the memory target) for the buffer cache, but it acquires (commits) only the required amount of physical memory for the current load. You can query the **bpool_commit_target** and **bpool_committed columns** in the sys.dm_os_sys_info catalog view to return the number of pages reserved as the memory target and the number of pages currently committed in the buffer cache, respectively.

The interval between SQL Server startup and when the buffer cache obtains its memory target is called ramp-up. During this time, read requests fill the buffers as needed. For example, a single 8 KB page read request fills a single buffer page. This means the ramp-up depends on the number and type of client requests. Ramp-up is expedited by transforming single page read requests into aligned eight page requests (making up one extent). This allows the ramp-up to finish much faster, especially on machines with a lot of memory. For more information about pages and extents, refer to Pages and Extents Architecture Guide.

Because the buffer manager uses most of the memory in the SQL Server process, it cooperates with the memory manager to allow other components to use its buffers. The buffer manager interacts primarily with the following components:

- Resource manager to control overall memory usage and, in 32-bit platforms, to control address space usage.
- Database manager and the SQL Server Operating System (SQLOS) for low-level file I/O operations.
- Log manager for write-ahead logging.

### Supported Features

The buffer manager supports the following features:

- The buffer manager is **non-uniform memory access (NUMA)** aware. Buffer cache pages are distributed across hardware NUMA nodes, which allows a thread to access a buffer page that is allocated on the local NUMA node rather than from foreign memory.
- The buffer manager supports **Hot Add Memory**, which allows users to add physical memory without restarting the server.
- The buffer manager supports **large pages** on 64-bit platforms. The page size is specific to the version of Windows.

> **NOTE**
>
> Prior to SQL Server 2012 (11.x), enabling large pages in SQL Server requires trace flag 834.

- The buffer manager provides additional diagnostics that are exposed through dynamic management views. You can use these views to monitor a variety of operating system resources that are specific to SQL Server. For example, you can use the sys.dm_os_buffer_descriptors view to monitor the pages in the buffer cache.

### Disk I/O

The buffer manager only performs reads and writes to the database. Other file and database operations such as open, close, extend, and shrink are performed by the database manager and file manager components.

Disk I/O operations by the buffer manager have the following characteristics:

- All I/Os are performed asynchronously, which allows the calling thread to continue processing while the I/O operation takes place in the background.
- All I/Os are issued in the calling threads unless the affinity I/O option is in use. The affinity I/O mask option binds SQL Server disk I/O to a specified subset of CPUs. In high-end SQL Server online transactional processing (OLTP) environments, this extension can enhance the performance of SQL Server threads issuing I/Os.
- Multiple page I/Os are accomplished with scatter-gather I/O, which allows data to be transferred into or out of noncontiguous areas of memory. This means that SQL Server can quickly fill or flush the buffer cache while avoiding multiple physical I/O requests.

#### Long I/O requests

The buffer manager reports on any I/O request that has been outstanding for at least 15 seconds. This helps the system administrator distinguish between SQL Server problems and I/O subsystem problems. Error message 833 is reported and appears in the SQL Server error log as follows:

```
SQL Server has encountered ## occurrence(s) of I/O requests taking longer than 15 seconds to complete on file
[##] in database [##] (#). The OS file handle is 0x00000. The offset of the latest long I/O is: 0x00000.
```

A long I/O may be either a read or a write; it is not currently indicated in the message. Long-I/O messages are warnings, not errors. They do not indicate problems with SQL Server but with the underlying I/O system. The messages are reported to help the system administrator find the cause of poor SQL Server response times more quickly, and to distinguish problems that are outside the control of SQL Server. As such, they do not require any action, but the system administrator should investigate why the I/O request took so long, and whether the time is justifiable.

#### Causes of Long-I/O Requests

A long-I/O message may indicate that an I/O is permanently blocked and will never complete (known as lost I/O), or merely that it just has not completed yet. It is not possible to tell from the message which scenario is the case, although a lost I/O will often lead to a latch timeout.

Long I/Os often indicate a SQL Server workload that is too intense for the disk subsystem. An inadequate disk subsystem may be indicated when:

- Multiple long I/O messages appear in the error log during a heavy SQL Server workload.
- Perfmon counters show long disk latencies, long disk queues, or no disk idle time.

Long I/Os may also be caused by a component in the I/O path (for example, a driver, controller, or firmware) continually postponing servicing an old I/O request in favor of servicing newer requests that are closer to the current position of the disk head. The common technique of processing requests in priority based upon which ones are closest to the current position of the read/write head is known as "elevator seeking." This may be difficult to corroborate with the Windows System Monitor (PERFMON.EXE) tool because most I/Os are being serviced promptly. Long I/O requests can be aggravated by workloads that perform large amounts of sequential I/O, such as backup and restore, table scans, sorting, creating indexes, bulk loads, and zeroing out files.

Isolated long I/Os that do not appear related to any of the previous conditions may be caused by a hardware or driver problem. The system event log may contain a related event that helps to diagnose the problem.

### Error Detection

Database pages can use one of two optional mechanisms that help insure the integrity of the page from the time it is written to disk until it is read again: torn page protection and checksum protection. These mechanisms allow an independent method of verifying the correctness of not only the data storage, but hardware components such as

controllers, drivers, cables, and even the operating system. The protection is added to the page just before writing it to disk, and verified after it is read from disk.

SQL Server will retry any read that fails with a checksum, torn page, or other I/O error four times. If the read is successful in any one of the retry attempts, a message will be written to the error log and the command that triggered the read will continue. If the retry attempts fail, the command will fail with error message 824.

The kind of page protection used is an attribute of the database containing the page. Checksum protection is the default protection for databases created in SQL Server 2005 and later. The page protection mechanism is specified at database creation time, and may be altered by using ALTER DATABASE SET. You can determine the current page protection setting by querying the *page_verify_option* column in the sys.databases catalog view or the *IsTornPageDetectionEnabled* property of the DATABASEPROPERTYEX function.

> **NOTE**
>
> If the page protection setting is changed, the new setting does not immediately affect the entire database. Instead, pages adopt the current protection level of the database whenever they are written next. This means that the database may be composed of pages with different kinds of protection.

**Torn Page Protection**

Torn page protection, introduced in SQL Server 2000, is primarily a way of detecting page corruptions due to power failures. For example, an unexpected power failure may leave only part of a page written to disk. When torn page protection is used, a specific 2-bit signature pattern for each 512-byte sector in the 8-kilobyte (KB) database page and stored in the database page header when the page is written to disk. When the page is read from disk, the torn bits stored in the page header are compared to the actual page sector information. The signature pattern alternates between binary 01 and 10 with every write, so it is always possible to tell when only a portion of the sectors made it to disk: if a bit is in the wrong state when the page is later read, the page was written incorrectly and a torn page is detected. Torn page detection uses minimal resources; however, it does not detect all errors caused by disk hardware failures. For information on setting torn page detection, see ALTER DATABASE SET Options (Transact-SQL).

**Checksum Protection**

Checksum protection, introduced in SQL Server 2005, provides stronger data integrity checking. A checksum is calculated for the data in each page that is written, and stored in the page header. Whenever a page with a stored checksum is read from disk, the database engine recalculates the checksum for the data in the page and raises error 824 if the new checksum is different from the stored checksum. Checksum protection can catch more errors than torn page protection because it is affected by every byte of the page, however, it is moderately resource intensive. When checksum is enabled, errors caused by power failures and flawed hardware or firmware can be detected any time the buffer manager reads a page from disk. For information on setting checksum, see ALTER DATABASE SET Options (Transact-SQL).

> **IMPORTANT**
>
> When a user or system database is upgraded to SQL Server 2005 or a later version, the PAGE_VERIFY value (NONE or TORN_PAGE_DETECTION) is retained. We recommend that you use CHECKSUM. TORN_PAGE_DETECTION may use fewer resources but provides a minimal subset of the CHECKSUM protection.

## Understanding Non-uniform Memory Access

SQL Server is non-uniform memory access (NUMA) aware, and performs well on NUMA hardware without special configuration. As clock speed and the number of processors increase, it becomes increasingly difficult to reduce the memory latency required to use this additional processing power. To circumvent this, hardware vendors provide large L3 caches, but this is only a limited solution. NUMA architecture provides a scalable solution to this

problem. SQL Server has been designed to take advantage of NUMA-based computers without requiring any application changes. For more information, see How to: Configure SQL Server to Use Soft-NUMA.

## See Also

Server Memory Server Configuration Options
Reading Pages
Writing Pages
How to: Configure SQL Server to Use Soft-NUMA
Requirements for Using Memory-Optimized Tables
Resolve Out Of Memory Issues Using Memory-Optimized Tables

# Reading Pages

5/3/2018 • 6 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

The I/O from an instance of the SQL Server Database Engine includes logical and physical reads. A logical read occurs every time the Database Engine requests a page from the buffer cache. If the page is not currently in the buffer cache, a physical read first copies the page from disk into the cache.

The read requests generated by an instance of the Database Engine are controlled by the relational engine and optimized by the storage engine. The relational engine determines the most effective access method (such as a table scan, an index scan, or a keyed read); the access methods and buffer manager components of the storage engine determine the general pattern of reads to perform, and optimize the reads required to implement the access method. The thread executing the batch schedules the reads.

## Read-Ahead

The Database Engine supports a performance optimization mechanism called read-ahead. Read-ahead anticipates the data and index pages needed to fulfill a query execution plan and brings the pages into the buffer cache before they are actually used by the query. This allows computation and I/O to overlap, taking full advantage of both the CPU and the disk.

The read-ahead mechanism allows the Database Engine to read up to 64 contiguous pages (512KB) from one file. The read is performed as a single scatter-gather read to the appropriate number of (probably non-contiguous) buffers in the buffer cache. If any of the pages in the range are already present in the buffer cache, the corresponding page from the read will be discarded when the read completes. The range of pages may also be "trimmed" from either end if the corresponding pages are already present in the cache.

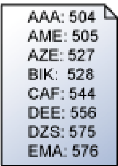
There are two kinds of read-ahead: one for data pages and one for index pages.

### Reading Data Pages

Table scans used to read data pages are very efficient in the Database Engine. The index allocation map (IAM) pages in a SQL Server database list the extents used by a table or index. The storage engine can read the IAM to build a sorted list of the disk addresses that must be read. This allows the storage engine to optimize its I/Os as large sequential reads that are performed in sequence, based on their location on the disk. For more information about IAM pages, see Managing Space Used by Objects.

### Reading Index Pages

The storage engine reads index pages serially in key order. For example, this illustration shows a simplified representation of a set of leaf pages that contains a set of keys and the intermediate index node mapping the leaf pages. For more information about the structure of pages in an index, see Clustered Index Structures.



The storage engine uses the information in the intermediate index page above the leaf level to schedule serial read-aheads for the pages that contain the keys. If a request is made for all the keys from ABC to DEF, the storage engine first reads the index page above the leaf page. However, it does not just read each data page in sequence from page 504 to page 556 (the last page with keys in the specified range). Instead, the storage engine scans the intermediate index page and builds a list of the leaf pages that must be read. The storage engine then schedules all the reads in key order. The storage engine also recognizes that pages 504/505 and 527/528 are contiguous and performs a single scatter read to retrieve the adjacent pages in a single operation. When there are many pages to be retrieved in a serial operation, the storage engine schedules a block of reads at a time. When a subset of these reads is completed, the storage engine schedules an equal number of new reads until all the required reads have been scheduled.

The storage engine uses prefetching to speed base table lookups from nonclustered indexes. The leaf rows of a nonclustered index contain pointers to the data rows that contain each specific key value. As the storage engine reads through the leaf pages of the nonclustered index, it also starts scheduling asynchronous reads for the data rows whose pointers have already been retrieved. This allows the storage engine to retrieve data rows from the underlying table before it has completed the scan of the nonclustered index. Prefetching is used regardless of whether the table has a clustered index. SQL Server Enterprise uses more prefetching than other editions of SQL Server, allowing more pages to be read ahead. The level of prefetching is not configurable in any edition. For more information about nonclustered indexes, see Nonclustered Index Structures.

## Advanced Scanning

In SQL Server Enterprise, the advanced scan feature allows multiple tasks to share full table scans. If the execution plan of a Transact-SQL statement requires a scan of the data pages in a table and the Database Engine detects that the table is already being scanned for another execution plan, the Database Engine joins the second scan to the first, at the current location of the second scan. The Database Engine reads each page one time and passes the rows from each page to both execution plans. This continues until the end of the table is reached.

At that point, the first execution plan has the complete results of a scan, but the second execution plan must still retrieve the data pages that were read before it joined the in-progress scan. The scan for the second execution plan then wraps back to the first data page of the table and scans forward to where it joined the first scan. Any number of scans can be combined like this. The Database Engine will keep looping through the data pages until it has completed all the scans. This mechanism is also called "merry-go-round scanning" and demonstrates why the order of the results returned from a SELECT statement cannot be guaranteed without an ORDER BY clause.

For example, assume that you have a table with 500,000 pages. UserA executes a Transact-SQL statement that requires a scan of the table. When that scan has processed 100,000 pages, UserB executes another Transact-SQL statement that scans the same table. The Database Engine schedules one set of read requests for pages after 100,001, and passes the rows from each page back to both scans. When the scan reaches the 200,000th page, UserC executes another Transact-SQL statement that scans the same table. Starting with page 200,001, the Database Engine passes the rows from each page it reads back to all three scans. After it reads the 500,000th row, the scan for UserA is complete, and the scans for UserB and UserC wrap back and start to read the pages starting with page 1. When the Database Engine gets to page 100,000, the scan for UserB is completed. The scan for UserC then keeps going alone until it reads page 200,000. At this point, all the scans have been completed.

Without advanced scanning, each user would have to compete for buffer space and cause disk arm contention. The same pages would then be read once for each user, instead of read one time and shared by multiple users, slowing down performance and taxing resources.

## See Also

Pages and Extents Architecture Guide
Writing Pages

# Writing Pages

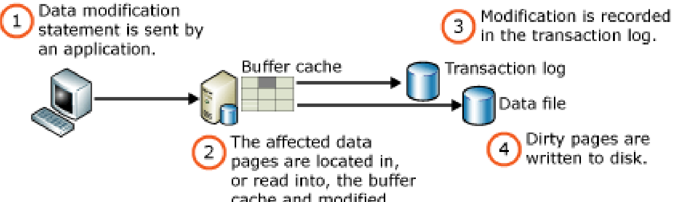
5/3/2018 • 3 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

The I/O from an instance of the Database Engine includes logical and physical writes. A logical write occurs when data is modified in a page in the buffer cache. A physical write occurs when the page is written from the buffer cache to disk.

When a page is modified in the buffer cache, it is not immediately written back to disk; instead, the page is marked as dirty. This means that a page can have more than one logical write made before it is physically written to disk. For each logical write, a transaction log record is inserted in the log cache that records the modification. The log records must be written to disk before the associated dirty page is removed from the buffer cache and written to disk. SQL Server uses a technique known as write-ahead logging that prevents writing a dirty page before the associated log record is written to disk. This is essential to the correct working of the recovery manager. For more information, see Write-Ahead Transaction Log.

The following illustration shows the process for writing a modified data page.




When the buffer manager writes a page, it searches for adjacent dirty pages that can be included in a single gather-write operation. Adjacent pages have consecutive page IDs and are from the same file; the pages do not have to be contiguous in memory. The search continues both forward and backward until one of the following events occurs:

- A clean page is found.
- 32 pages have been found.
- A dirty page is found whose log sequence number (LSN) has not yet been flushed in the log.
- A page is found that cannot be immediately latched.

In this way, the entire set of pages can be written to disk with a single gather-write operation.

Just before a page is written, the form of page protection specified in the database is added to the page. If torn page protection is added, the page must be latched EX(clusively) for the I/O. This is because the torn page protection modifies the page, making it unsuitable for any other thread to read. If checksum page protection is added, or the database uses no page protection, the page is latched with an UP(date) latch for the I/O. This latch prevents anyone else from modifying the page during the write, but still allows readers to use it. For more information about disk I/O page protection options, see Buffer Management.

A dirty page is written to disk in one of three ways:

- Lazy writing
  The lazy writer is a system process that keeps free buffers available by removing infrequently used pages from the buffer cache. Dirty pages are first written to disk.

- Eager writing
  The eager write process writes dirty data pages associated with nonlogged operations such as bulk insert

and select into. This process allows creating and writing new pages to take place in parallel. That is, the calling operation does not have to wait until the entire operation finishes before writing the pages to disk.

- Checkpoint
  The checkpoint process periodically scans the buffer cache for buffers with pages from a specified database and writes all dirty pages to disk. Checkpoints save time during a later recovery by creating a point at which all dirty pages are guaranteed to have been written to disk. The user may request a checkpoint operation by using the CHECKPOINT command, or the Database Engine may generate automatic checkpoints based on the amount of log space used and time elapsed since the last checkpoint. In addition, a checkpoint is generated when certain activities occur. For example, when a data or log file is added or removed from a database, or when the instance of SQL Server is stopped. For more information, see Checkpoints and the Active Portion of the Log.

The lazy writing, eager writing, and checkpoint processes do not wait for the I/O operation to complete. They always use asynchronous (or overlapped) I/O and continue with other work, checking for I/O success later. This allows SQL Server to maximize both CPU and I/O resources for the appropriate tasks.

## See Also

Pages and Extents Architecture Guide
Reading Pages

# Pages and Extents Architecture Guide

5/3/2018 • 13 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

The page is the fundamental unit of data storage in SQL Server. An extent is a collection of eight physically contiguous pages. Extents help efficiently manage pages. This guide describes the data structures that are used to manage pages and extents in all versions of SQL Server. Understanding the architecture of pages and extents is important for designing and developing databases that perform efficiently.

## Pages and Extents

The fundamental unit of data storage in SQL Server is the page. The disk space allocated to a data file (.mdf or .ndf) in a database is logically divided into pages numbered contiguously from 0 to n. 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.

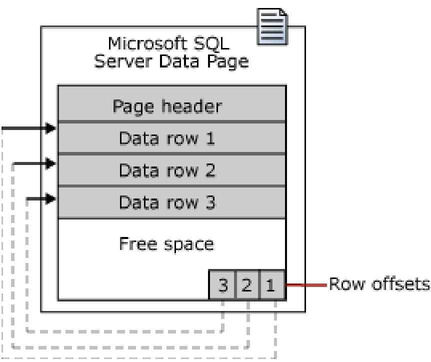
| PAGE TYPE | CONTENTS |
| --- | --- |
| Data | Data rows with all data, except text, ntext, image, nvarchar(max), varchar(max), varbinary(max), and xml data, when text in row is set to ON. |
| Index | Index entries. |
| Text/Image | Large object data types: (text, ntext, image, nvarchar(max), varchar(max), varbinary(max), and xml data)<br>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. |

| PAGE TYPE | CONTENTS |
| --- | --- |
| 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.

Data rows are put on the page serially, starting immediately after the header. A row offset table starts at the end of the page, and each row offset table contains one entry for each row on the page. Each entry records how far the first byte of the row is from the start of 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**

Rows cannot span pages, however portions of the row may be moved off the row's page so that the row can actually be very large. The maximum amount of data and overhead that is contained in a single row on a page is 8,060 bytes (8 KB). However, this does not include the data stored in the Text/Image page type.

This restriction is relaxed for tables that contain varchar, nvarchar, varbinary, or sql_variant columns. 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.

This is done whenever an insert or update operation increases the total size of the row beyond the 8060 byte limit. 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.

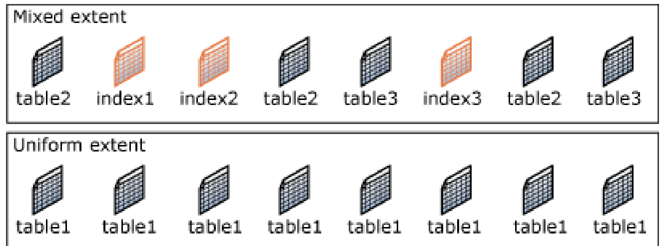
### Extents

Extents are the basic unit in which space is managed. An extent is eight physically contiguous pages, or 64 KB. This means SQL Server databases have 16 extents per megabyte.

To make its space allocation efficient, SQL Server does not allocate whole extents to tables with small amounts of data. 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 and Free Space

The SQL Server data structures that manage extent allocations and track free space have a relatively simple structure. This has the following benefits:

- The free space information is densely packed, so relatively few pages contain this information.
  This increases speed by reducing the amount of disk reads that are required to retrieve allocation information. This also increases the chance that the allocation pages will remain in memory and not require more reads.
- Most of the allocation information is not chained together. This simplifies the maintenance of the allocation information.
  Each page allocation or deallocation can be performed quickly. This decreases the contention between concurrent tasks having to allocate or deallocate pages.

**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 also 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.

Each extent has the following bit patterns set in the GAM and SGAM, based on its current use.

| CURRENT USE OF EXTENT | GAM BIT SETTING | SGAM BIT SETTING |
| --- | --- | --- |
| Free, not being used | 1 | 0 |
| Uniform extent, or full mixed extent | 0 | 0 |
| Mixed extent with free pages | 0 | 1 |

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

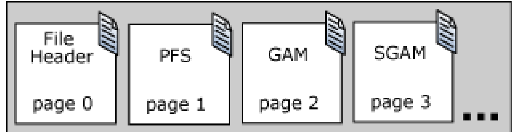
SGAM bit is set to 0. The algorithms that are actually used internally by the SQL Server Database Engine are more sophisticated than what is described in this topic, because the SQL Server Database Engine distributes data evenly in a database. However, even the real algorithms are simplified by not having to manage chains of extent allocation information.

**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.

After an extent has been allocated to an object, the Database Engine uses the PFS pages to record which pages in the extent are allocated or free. This information is used when the Database Engine has to allocate a new page. The amount of free space in a page is only maintained for heap and Text/Image pages. It is used when the Database Engine has to find a page with free space available to hold a newly inserted row. Indexes do not require that the page free space be tracked, because the point at which to insert a new row is set by the index key values.

A PFS page is the first page after the file header page in a data file (page id 1). This is followed by a GAM page (page id 2), and then an SGAM page (page id 3). There is a PFS page approximately 8,000 pages in size after the first PFS page. There is another GAM page 64,000 extents after the first GAM page on page 2, and another SGAM page 64,000 extents after the first SGAM page on page 3. The following illustration shows the sequence of pages used by the SQL Server Database Engine to allocate and manage extents.



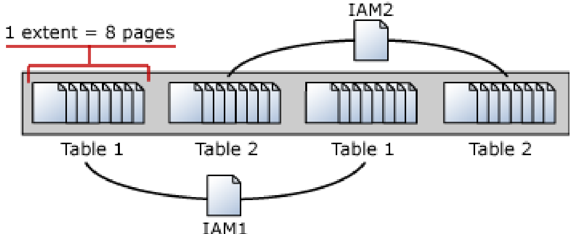

# Managing space used by objects

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.

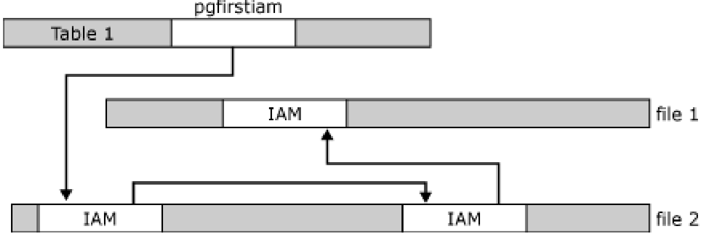
Each partition of a heap or index contains at least an IN_ROW_DATA allocation unit. It may also contain a LOB_DATA or ROW_OVERFLOW_DATA allocation unit, depending on the heap or index schema. For more information about allocation units, see Table and Index Organization.

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. Therefore, each allocation unit has at least one IAM page for each file on which it has extents. There may also be more than one IAM page on a file, if the range of the extents on the file allocated to the allocation unit exceeds the range that a single IAM page can record.




IAM pages are allocated as required for each allocation unit and are located randomly in the file. The system view, sys.system_internals_allocation_units, points to the first IAM page for an allocation unit. All the IAM pages for that allocation unit are linked in a chain.

> **IMPORTANT**
> The sys.system_internals_allocation_units system view is for internal use only and is subject to change. Compatibility is not guaranteed.




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 lots of 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. For indexes, the insertion point of a new row is set by the index key. In this case, the search process previously described does not occur.

The SQL Server Database Engine allocates a new extent to an allocation unit only when it cannot quickly find a page in an existing extent with sufficient space to hold the row being inserted.

The SQL Server Database Engine allocates extents from those available in the filegroup using a **proportional fill allocation algorithm**. If in the same filegroup with two files, one file has two times the free space as the other, two pages will be allocated from the file with the available space for every one page allocated from the other file. This means that every file in a filegroup should have a similar percentage of space used.

## 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. Differential backups read just the DCM pages to determine which extents have been modified. This greatly reduces the number of pages that a differential backup must scan. The length of time that a differential backup runs is proportional to the number of extents modified since the last BACKUP DATABASE statement and not the overall size of the database.

- **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. Although BCM pages appear in all databases, they are only relevant when the database is using the bulk-logged recovery model. In this recovery model, when a `BACKUP LOG` is performed, the backup process scans the BCMs for extents that have been modified. It then includes those extents in the log backup. This lets the bulk logged operations be recovered if the database is restored from a database backup and a sequence of transaction log backups. BCM pages are not relevant in a database that is using the simple recovery model, because no bulk logged operations are logged. They are not relevant in a database that is using the full recovery model, because that recovery model treats bulk logged operations as fully logged operations.

The interval between DCM pages and BCM pages is the same as the interval between GAM and SGAM page, 64,000 extents. The DCM and BCM pages are located behind the GAM and SGAM pages in a physical file:

# Post-migration Validation and Optimization Guide

5/3/2018 • 6 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

SQL Server post migration step is very crucial for reconciling any data accuracy and completeness, as well as uncover performance issues with the workload.

# Common Performance Scenarios

Below are some of the common performance scenarios encountered after migrating to SQL Server Platform and how to resolve them. These include scenarios that are specific to SQL Server to SQL Server migration (older versions to newer versions), as well as foreign platform (such as Oracle, DB2, MySQL and Sybase) to SQL Server migration.

## Query regressions due to change in CE version

**Applies to:** SQL Server to SQL Server migration.

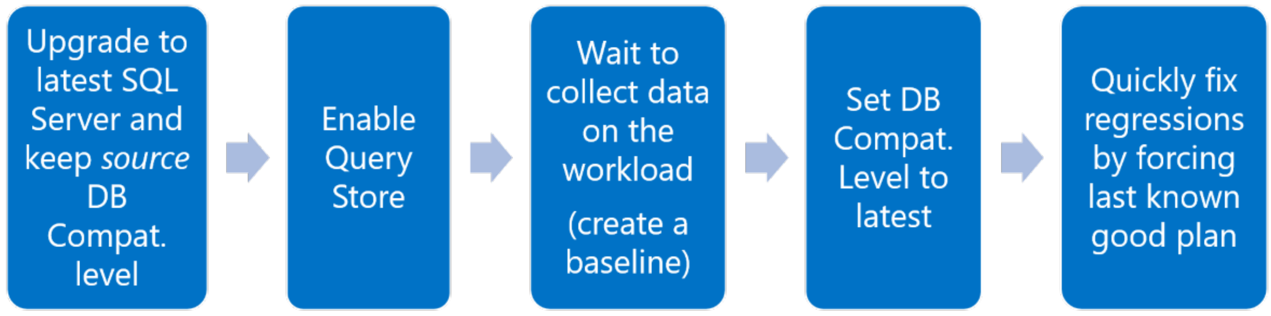
When migrating from an older versions of SQL Server to SQL Server 2014 (12.x) or newer, and upgrading the database compatibility level to the latest one, a workload may be exposed to the risk of performance regression.

This is because starting with SQL Server 2014 (12.x), all Query Optimizer changes are tied to the latest database compatibility level, so plans are not changed right at point of upgrade but rather when a user changes the `COMPATIBILITY_LEVEL` database option to the latest one. This capability, in combination with Query Store gives you a great level of control over the query performance in the upgrade process.

For more information on Query Optimizer changes introduced in SQL Server 2014 (12.x), see Optimizing Your Query Plans with the SQL Server 2014 Cardinality Estimator.

#### Steps to resolve

Change the database compatibility level to the source version, and follow the recommended upgrade workflow as shown in the following picture:




For more information on this topic, see Keep performance stability during the upgrade to newer SQL Server.

## Sensitivity to parameter sniffing

**Applies to:** Foreign platform (such as Oracle, DB2, MySQL and Sybase) to SQL Server migration.

**NOTE**

For SQL Server to SQL Server migrations, if this issue existed in the source SQL Server, migrating to a newer version of SQL Server as-is will not address this scenario.

SQL Server compiles query plans on stored procedures by using sniffing the input parameters at the first compile, generating a parameterized and reusable plan, optimized for that input data distribution. Even if not stored procedures, most statements generating trivial plans will be parameterized. After a plan is first cached, any future execution maps to a previously cached plan. A potential problem arises when that first compilation may not have used the most common sets of parameters for the usual workload. For different parameters, the same execution plan becomes inefficient. For more information on this topic, see Parameter Sniffing.

**Steps to resolve**

1. Use the `RECOMPILE` hint. A plan is calculated every time adapted to each parameter value.
2. Rewrite the stored procedure to use the option `(OPTIMIZE FOR(<input parameter> = <value>))`. Decide which value to use that suits most of the relevant workload, creating and maintaining one plan that becomes efficient for the parameterized value.
3. Rewrite the stored procedure using local variable inside the procedure. Now the optimizer uses the density vector for estimations, resulting in the same plan regardless of the parameter value.
4. Rewrite the stored procedure to use the option `(OPTIMIZE FOR UNKNOWN)`. Same effect as using the local variable technique.
5. Rewrite the query to use the hint `DISABLE_PARAMETER_SNIFFING`. Same effect as using the local variable technique by totally disabling parameter sniffing, unless `OPTION(RECOMPILE)`, `WITH RECOMPILE` or `OPTIMIZE FOR <value>` is used.

**TIP**

Leverage the Management Studio Plan Analysis feature to quickly identify if this is an issue. More information available here.

## Missing indexes

**Applies to:** Foreign platform (such as Oracle, DB2, MySQL and Sybase) and SQL Server to SQL Server migration.

Incorrect or missing indexes causes extra I/O that leads to extra memory and CPU being wasted. This maybe because workload profile has changed such as using different predicates, invalidating existing index design. Evidence of a poor indexing strategy or changes in workload profile include:

- Look for duplicate, redundant, rarely used and completely unused indexes.
- Special care with unused indexes with updates.

**Steps to resolve**

1. Leverage the graphical execution plan for any Missing Index references.
2. Indexing suggestions generated by Database Engine Tuning Advisor.
3. Leverage the Missing Indexes DMV or through the SQL Server Performance Dashboard.
4. Leverage pre-existing scripts that can use existing DMVs to provide insight into any missing, duplicate, redundant, rarely used and completely unused indexes, but also if any index reference is hinted/hard-coded into existing procedures and functions in your database.

**TIP**

Examples of such pre-existing scripts include Index Creation and Index Information.

## Inability to use predicates to filter data

**Applies to:** Foreign platform (such as Oracle, DB2, MySQL and Sybase) and SQL Server to SQL Server migration.

**NOTE**

For SQL Server to SQL Server migrations, if this issue existed in the source SQL Server, migrating to a newer version of SQL Server as-is will not address this scenario.

SQL Server Query Optimizer can only account for information that is known at compile time. If a workload relies on predicates that can only be known at execution time, then the potential for a poor plan choice increases. For a better-quality plan, predicates must be **SARGable**, or **S**earch **Arg**ument**able**.

Some examples of non-SARGable predicates:

- Implicit data conversions, like VARCHAR to NVARCHAR, or INT to VARCHAR. Look for runtime CONVERT_IMPLICIT warnings in the Actual Execution Plans. Converting from one type to another can also cause a loss of precision.
- Complex undetermined expressions such as `WHERE UnitPrice + 1 < 3.975`, but not `WHERE UnitPrice < 320 * 200 * 32`.
- Expressions using functions, such as `WHERE ABS(ProductID) = 771` or `WHERE UPPER(LastName) = 'Smith'`
- Strings with a leading wildcard character, such as `WHERE LastName LIKE '%Smith'`, but not `WHERE LastName LIKE 'Smith%'`.

**Steps to resolve**

1. Always declare variables/parameters as the intended target data type.
   - This may involve comparing any user-defined code construct that is stored in the database (such as stored procedures, user-defined functions or views) with system tables that hold information on data types used in underlying tables (such as sys.columns).
2. If unable to traverse all code to the previous point, then for the same purpose, change the data type on the table to match any variable/parameter declaration.
3. Reason out the usefulness of the following constructs:
   - Functions being used as predicates;
   - Wildcard searches;
   - Complex expressions based on columnar data – evaluate the need to instead create persisted computed columns, which can be indexed;

**NOTE**

All of the above can be done programmatically.

## Use of Table Valued Functions (Multi-Statement vs Inline)

**Applies to:** Foreign platform (such as Oracle, DB2, MySQL and Sybase) and SQL Server to SQL Server migration.

> **NOTE**
>
> For SQL Server to SQL Server migrations, if this issue existed in the source SQL Server, migrating to a newer version of SQL Server as-is will not address this scenario.

Table Valued Functions return a table data type that can be an alternative to views. While views are limited to a single `SELECT` statement, user-defined functions can contain additional statements that allow more logic than is possible in views.

> **IMPORTANT**
>
> Since the output table of an MSTVF (Multi-Statement Table Valued Function) is not created at compile time, the SQL Server Query Optimizer relies on heuristics, and not actual statistics, to determine row estimations. Even if indexes are added to the base table(s), this is not going to help. For MSTVFs, SQL Server uses a fixed estimation of 1 for the number of rows expected to be returned by an MSTVF (starting with SQL Server 2014 (12.x) that fixed estimation is 100 rows).

**Steps to resolve**

1. If the Multi-Statement TVF is single statement only, convert to Inline TVF.

```
CREATE FUNCTION dbo.tfnGetRecentAddress(@ID int)
RETURNS @tblAddress TABLE
([Address] VARCHAR(60) NOT NULL)
AS
BEGIN
  INSERT INTO @tblAddress ([Address])
  SELECT TOP 1 [AddressLine1]
  FROM [Person].[Address]
  WHERE  AddressID = @ID
  ORDER BY [ModifiedDate] DESC
RETURN
END
```

To

```
CREATE FUNCTION dbo.tfnGetRecentAddress_inline(@ID int)
RETURNS TABLE
AS
RETURN (
  SELECT TOP 1 [AddressLine1] AS [Address]
  FROM [Person].[Address]
  WHERE  AddressID = @ID
  ORDER BY [ModifiedDate] DESC
)
```

2. If more complex, consider using intermediate results stored in Memory-Optimized tables or temporary tables.

# Additional Reading

Best Practice with the Query Store
Memory-Optimized Tables
User-Defined Functions
Table Variables and Row Estimations - Part 1
Table Variables and Row Estimations - Part 2
Execution Plan Caching and Reuse

# Performance Center for SQL Server Database Engine and Azure SQL Database

5/3/2018 • 2 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

This page provides links to help you locate the information that you need about performance in the SQL Server Database Engine and Azure SQL Database.

### Legend

indicates that the feature is available only in SQL Server Database Engine (both SQL Server running on-premises and SQL Server running in an Azure Virtual Machine).

indicates that the feature is available only in Azure SQL Database.

indicates that the feature is available in both SQL Server and SQL Database.

## Configuration Options for Performance

SQL Server provides the ability to affect database engine performance through a number of configuration options at the SQL Server Database Engine level. With Azure SQL Database, Microsoft performs most, but not all, of these optimizations for you.

| | |
|---|---|
| **Disk configuration options** | Disk striping and RAID |
| **Data and log file configuration options** | Place Data and Log Files on Separate Drives<br>View or Change the Default Locations for Data and Log Files (SQL Server Management Studio) |
| **TempDB configuration options** | Performance Improvements in TempDB<br>Database Engine Configuration - TempDB<br>Using SSDs in Azure VMs to store SQL Server TempDB and Buffer Pool Extensions<br>Disk and performance best practices for temporary disk for SQL Server in Azure Virtual Machines |

| | |
|---|---|
| **Server Configuration Options** | • **Processor configuration options**<br>◦ affinity mask Server Configuration Option<br>◦ affinity Input-Output mask Server Configuration Option<br>◦ affinity64 mask Server Configuration Option<br>◦ affinity64 Input-Output mask Server Configuration Option<br>◦ Configure the max worker threads Server Configuration Option<br>• **Memory configuration options**<br>◦ Server Memory Server Configuration Options<br>• **Index configuration options**<br>◦ Configure the fill factor Server Configuration Option<br>• **Query configuration options**<br>◦ Configure the min memory per query Server Configuration Option<br>◦ Configure the query governor cost limit Server Configuration Option<br>◦ Configure the max degree of parallelism Server Configuration Option<br>◦ Configure the cost threshold for parallelism Server Configuration Option<br>◦ optimize for ad hoc workloads Server Configuration Option<br>• **Backup configuration options**<br>◦ View or Configure the backup compression default Server Configuration Option |
| **Database configuration optimization options** | Data Compression<br>View or Change the Compatibility Level of a Database<br>ALTER DATABASE SCOPED CONFIGURATION (Transact-SQL) |
| **Table configuration optimization** | Partitioned Tables and Indexes |
| **Database Engine Performance in an Azure Virtual Machine** | Quick check list<br>Virtual machine size and storage account considerations<br>Disks and performance considerations<br>I/O Performance Considerations<br>Feature specific performance considerations |

# Query Performance Options

| | |
|---|---|

| | |
|---|---|
| **Indexes** | Reorganize and Rebuild Indexes<br>Specify Fill Factor for an Index<br>Configure Parallel Index Operations<br>SORT_IN_TEMPDB Option For Indexes<br>Improve the Performance of Full-Text Indexes<br>Configure the min memory per query Server Configuration Option<br>Configure the index create memory Server Configuration Option |
| **Partitioned Tables and Indexes** | Benefits of Partitioning |
| **Joins** | Join Fundamentals<br>Nested Loops join<br>Merge join<br>Hash join |
| **Subqueries** | Subquery Fundamentals<br>Correlated subqueries<br>Subquery types |
| **Stored Procedures** | CREATE PROCEDURE (Transact-SQL) |
| **User-Defined Functions** | CREATE FUNCTION (Transact-SQL) |
| **Parallelism optimization** | Configure the max worker threads Server Configuration Option<br>ALTER DATABASE SCOPED CONFIGURATION (Transact-SQL) |
| **Query optimizer optimization** | ALTER DATABASE SCOPED CONFIGURATION (Transact-SQL) |
| **Statistics** | When to Update Statistics<br>Update Statistics |
| **In-Memory OLTP (In-Memory Optimization)** | Memory-Optimized Tables<br>Natively Compiled Stored Procedures<br>Creating and Accessing Tables in TempDB from Natively Compiled Stored Procedures<br>Troubleshooting Common Performance Problems with Memory-Optimized Hash Indexes<br>Demonstration: Performance Improvement of In-Memory OLTP |

## See Also

Monitor and Tune for Performance
Monitoring Performance By Using the Query Store
Azure SQL Database performance guidance for single databases
Optimizing Azure SQL Database Performance using Elastic Pools
Azure Query Performance Insight
Index Design Guide
Memory Management Architecture Guide
Pages and Extents Architecture Guide
Post-migration Validation and Optimization Guide

Query Processing Architecture Guide
SQL Server Transaction Locking and Row Versioning Guide
SQL Server Transaction Log Architecture and Management Guide
Thread and Task Architecture Guide

# Query Processing Architecture Guide

5/3/2018 • 72 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

The SQL Server Database Engine processes queries on various data storage architectures such as local tables, partitioned tables, and tables distributed across multiple servers. The following topics cover how SQL Server processes queries and optimizes query reuse through execution plan caching.

## SQL Statement Processing

Processing a single SQL statement is the most basic way that SQL Server executes SQL statements. The steps used to process a single `SELECT` statement that references only local base tables (no views or remote tables) illustrates the basic process.

#### Logical Operator Precedence

When more than one logical operator is used in a statement, `NOT` is evaluated first, then `AND`, and finally `OR`. Arithmetic, and bitwise, operators are handled before logical operators. For more information, see Operator Precedence.

In the following example, the color condition pertains to product model 21, and not to product model 20, because `AND` has precedence over `OR`.

```
SELECT ProductID, ProductModelID
FROM Production.Product
WHERE ProductModelID = 20 OR ProductModelID = 21
  AND Color = 'Red';
GO
```

You can change the meaning of the query by adding parentheses to force evaluation of the `OR` first. The following query finds only products under models 20 and 21 that are red.

```
SELECT ProductID, ProductModelID
FROM Production.Product
WHERE (ProductModelID = 20 OR ProductModelID = 21)
  AND Color = 'Red';
GO
```

Using parentheses, even when they are not required, can improve the readability of queries, and reduce the chance of making a subtle mistake because of operator precedence. There is no significant performance penalty in using parentheses. The following example is more readable than the original example, although they are syntactically the same.

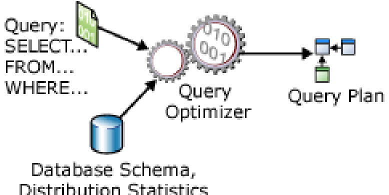
```
SELECT ProductID, ProductModelID
FROM Production.Product
WHERE ProductModelID = 20 OR (ProductModelID = 21
  AND Color = 'Red');
GO
```

#### Optimizing SELECT statements

A `SELECT` statement is non-procedural; it does not state the exact steps that the database server should use to

retrieve the requested data. This means that the database server must analyze the statement to determine the most efficient way to extract the requested data. This is referred to as optimizing the `SELECT` statement. The component that does this is called the Query Optimizer. The input to the Query Optimizer consists of the query, the database schema (table and index definitions), and the database statistics. The output of the Query Optimizer is a query execution plan, sometimes referred to as a query plan or just a plan. The contents of a query plan are described in more detail later in this topic.

The inputs and outputs of the Query Optimizer during optimization of a single `SELECT` statement are illustrated in the following diagram:




A `SELECT` statement defines only the following:

- The format of the result set. This is specified mostly in the select list. However, other clauses such as `ORDER BY` and `GROUP BY` also affect the final form of the result set.
- The tables that contain the source data. This is specified in the `FROM` clause.
- How the tables are logically related for the purposes of the `SELECT` statement. This is defined in the join specifications, which may appear in the `WHERE` clause or in an `ON` clause following `FROM`.
- The conditions that the rows in the source tables must satisfy to qualify for the `SELECT` statement. These are specified in the `WHERE` and `HAVING` clauses.

A query execution plan is a definition of the following:

- The sequence in which the source tables are accessed.
  Typically, there are many sequences in which the database server can access the base tables to build the result set. For example, if the `SELECT` statement references three tables, the database server could first access `TableA`, use the data from `TableA` to extract matching rows from `TableB`, and then use the data from `TableB` to extract data from `TableC`. The other sequences in which the database server could access the tables are:

  `TableC`, `TableB`, `TableA`, or
  `TableB`, `TableA`, `TableC`, or
  `TableB`, `TableC`, `TableA`, or
  `TableC`, `TableA`, `TableB`

- The methods used to extract data from each table.
  Generally, there are different methods for accessing the data in each table. If only a few rows with specific key values are required, the database server can use an index. If all the rows in the table are required, the database server can ignore the indexes and perform a table scan. If all the rows in a table are required but there is an index whose key columns are in an `ORDER BY`, performing an index scan instead of a table scan may save a separate sort of the result set. If a table is very small, table scans may be the most efficient method for almost all access to the table.

The process of selecting one execution plan from potentially many possible plans is referred to as optimization. The Query Optimizer is one of the most important components of a SQL database system. While some overhead is used by the Query Optimizer to analyze the query and select a plan, this overhead is typically saved several-fold when the Query Optimizer picks an efficient execution plan. For example, two construction companies can be given identical blueprints for a house. If one company spends a few days at the beginning to plan how they will build the house, and the other company begins building without planning, the company that takes the time to plan their project will probably finish first.

The SQL Server Query Optimizer is a cost-based Query Optimizer. Each possible execution plan has an associated cost in terms of the amount of computing resources used. The Query Optimizer must analyze the possible plans and choose the one with the lowest estimated cost. Some complex `SELECT` statements have thousands of possible execution plans. In these cases, the Query Optimizer does not analyze all possible combinations. Instead, it uses complex algorithms to find an execution plan that has a cost reasonably close to the minimum possible cost.

The SQL Server Query Optimizer does not choose only the execution plan with the lowest resource cost; it chooses the plan that returns results to the user with a reasonable cost in resources and that returns the results the fastest. For example, processing a query in parallel typically uses more resources than processing it serially, but completes the query faster. The SQL Server Query Optimizer will use a parallel execution plan to return results if the load on the server will not be adversely affected.

The SQL Server Query Optimizer relies on distribution statistics when it estimates the resource costs of different methods for extracting information from a table or index. Distribution statistics are kept for columns and indexes. They indicate the selectivity of the values in a particular index or column. For example, in a table representing cars, many cars have the same manufacturer, but each car has a unique vehicle identification number (VIN). An index on the VIN is more selective than an index on the manufacturer. If the index statistics are not current, the Query Optimizer may not make the best choice for the current state of the table. For more information about keeping index statistics current, see Statistics.

The SQL Server Query Optimizer is important because it enables the database server to adjust dynamically to changing conditions in the database without requiring input from a programmer or database administrator. This enables programmers to focus on describing the final result of the query. They can trust that the SQL Server Query Optimizer will build an efficient execution plan for the state of the database every time the statement is run.

**Processing a SELECT Statement**

The basic steps that SQL Server uses to process a single SELECT statement include the following:

1. The parser scans the `SELECT` statement and breaks it into logical units such as keywords, expressions, operators, and identifiers.
2. A query tree, sometimes referred to as a sequence tree, is built describing the logical steps needed to transform the source data into the format required by the result set.
3. The Query Optimizer analyzes different ways the source tables can be accessed. It then selects the series of steps that returns the results fastest while using fewer resources. The query tree is updated to record this exact series of steps. The final, optimized version of the query tree is called the execution plan.
4. The relational engine starts executing the execution plan. As the steps that require data from the base tables are processed, the relational engine requests that the storage engine pass up data from the rowsets requested from the relational engine.
5. The relational engine processes the data returned from the storage engine into the format defined for the result set and returns the result set to the client.

**Processing Other Statements**

The basic steps described for processing a `SELECT` statement apply to other SQL statements such as `INSERT`, `UPDATE`, and `DELETE`. `UPDATE` and `DELETE` statements both have to target the set of rows to be modified or deleted. The process of identifying these rows is the same process used to identify the source rows that contribute to the result set of a `SELECT` statement. The `UPDATE` and `INSERT` statements may both contain embedded `SELECT statements that provide the data values to be updated or inserted.

Even Data Definition Language (DDL) statements, such as `CREATE PROCEDURE` or `ALTER TABL`E, are ultimately resolved to a series of relational operations on the system catalog tables and sometimes (such as `ALTER TABLE ADD COLUMN`) against the data tables.

### Worktables

The relational engine may need to build a worktable to perform a logical operation specified in an SQL statement. Worktables are internal tables that are used to hold intermediate results. Worktables are generated for certain

`GROUP BY`, `ORDER BY`, or `UNION` queries. For example, if an `ORDER BY` clause references columns that are not covered by any indexes, the relational engine may need to generate a worktable to sort the result set into the order requested. Worktables are also sometimes used as spools that temporarily hold the result of executing a part of a query plan. Worktables are built in `tempdb` and are dropped automatically when they are no longer needed.

**View Resolution**

The SQL Server query processor treats indexed and nonindexed views differently:

- The rows of an indexed view are stored in the database in the same format as a table. If the Query Optimizer decides to use an indexed view in a query plan, the indexed view is treated the same way as a base table.
- Only the definition of a nonindexed view is stored, not the rows of the view. The Query Optimizer incorporates the logic from the view definition into the execution plan it builds for the SQL statement that references the nonindexed view.

The logic used by the SQL Server Query Optimizer to decide when to use an indexed view is similar to the logic used to decide when to use an index on a table. If the data in the indexed view covers all or part of the SQL statement, and the Query Optimizer determines that an index on the view is the low-cost access path, the Query Optimizer will choose the index regardless of whether the view is referenced by name in the query.

When an SQL statement references a nonindexed view, the parser and Query Optimizer analyze the source of both the SQL statement and the view and then resolve them into a single execution plan. There is not one plan for the SQL statement and a separate plan for the view.

For example, consider the following view:

```
USE AdventureWorks2014;
GO
CREATE VIEW EmployeeName AS
SELECT h.BusinessEntityID, p.LastName, p.FirstName
FROM HumanResources.Employee AS h
JOIN Person.Person AS p
ON h.BusinessEntityID = p.BusinessEntityID;
GO
```

Based on this view, both of these SQL statements perform the same operations on the base tables and produce the same results:

```
/* SELECT referencing the EmployeeName view. */
SELECT LastName AS EmployeeLastName, SalesOrderID, OrderDate
FROM AdventureWorks2014.Sales.SalesOrderHeader AS soh
JOIN AdventureWorks2014.dbo.EmployeeName AS EmpN
ON (soh.SalesPersonID = EmpN.BusinessEntityID)
WHERE OrderDate > '20020531';

/* SELECT referencing the Person and Employee tables directly. */
SELECT LastName AS EmployeeLastName, SalesOrderID, OrderDate
FROM AdventureWorks2014.HumanResources.Employee AS e
JOIN AdventureWorks2014.Sales.SalesOrderHeader AS soh
ON soh.SalesPersonID = e.BusinessEntityID
JOIN AdventureWorks2014.Person.Person AS p
ON e.BusinessEntityID =p.BusinessEntityID
WHERE OrderDate > '20020531';
```

The SQL Server Management Studio Showplan feature shows that the relational engine builds the same execution plan for both of these `SELECT` statements.

**Using Hints with Views**

Hints that are placed on views in a query may conflict with other hints that are discovered when the view is

expanded to access its base tables. When this occurs, the query returns an error. For example, consider the following view that contains a table hint in its definition:

```
USE AdventureWorks2014;
GO
CREATE VIEW Person.AddrState WITH SCHEMABINDING AS
SELECT a.AddressID, a.AddressLine1,
    s.StateProvinceCode, s.CountryRegionCode
FROM Person.Address a WITH (NOLOCK), Person.StateProvince s
WHERE a.StateProvinceID = s.StateProvinceID;
```

Now suppose you enter this query:

```
SELECT AddressID, AddressLine1, StateProvinceCode, CountryRegionCode
FROM Person.AddrState WITH (SERIALIZABLE)
WHERE StateProvinceCode = 'WA';
```

The query fails, because the hint `SERIALIZABLE` that is applied on view `Person.AddrState` in the query is propagated to both tables `Person.Address` and `Person.StateProvince` in the view when it is expanded. However, expanding the view also reveals the `NOLOCK` hint on `Person.Address`. Because the `SERIALIZABLE` and `NOLOCK` hints conflict, the resulting query is incorrect.

The `PAGLOCK`, `NOLOCK`, `ROWLOCK`, `TABLOCK`, or `TABLOCKX` table hints conflict with each other, as do the `HOLDLOCK`, `NOLOCK`, `READCOMMITTED`, `REPEATABLEREAD`, `SERIALIZABLE` table hints.

Hints can propagate through levels of nested views. For example, suppose a query applies the `HOLDLOCK` hint on a view `v1`. When `v1` is expanded, we find that view `v2` is part of its definition. `v2`'s definition includes a `NOLOCK` hint on one of its base tables. But this table also inherits the `HOLDLOCK` hint from the query on view `v1`. Because the `NOLOCK` and `HOLDLOCK` hints conflict, the query fails.

When the `FORCE ORDER` hint is used in a query that contains a view, the join order of the tables within the view is determined by the position of the view in the ordered construct. For example, the following query selects from three tables and a view:

```
SELECT * FROM Table1, Table2, View1, Table3
WHERE Table1.Col1 = Table2.Col1
    AND Table2.Col1 = View1.Col1
    AND View1.Col2 = Table3.Col2;
OPTION (FORCE ORDER);
```

And `View1` is defined as shown in the following:

```
CREATE VIEW View1 AS
SELECT Colx, Coly FROM TableA, TableB
WHERE TableA.ColZ = TableB.Colz;
```

The join order in the query plan is `Table1`, `Table2`, `TableA`, `TableB`, `Table3`.

**Resolving Indexes on Views**

As with any index, SQL Server chooses to use an indexed view in its query plan only if the Query Optimizer determines it is beneficial to do so.

Indexed views can be created in any edition of SQL Server. In some editions of some versions of SQL Server, the Query Optimizer automatically considers the indexed view. In some editions of some versions of SQL Server, to use an indexed view, the `NOEXPAND` table hint must be used. For clarification, see the documentation for each

version.

The SQL Server Query Optimizer uses an indexed view when the following conditions are met:

- These session options are set to `ON`:
  - `ANSI_NULLS`
  - `ANSI_PADDING`
  - `ANSI_WARNINGS`
  - `ARITHABORT`
  - `CONCAT_NULL_YIELDS_NULL`
  - `QUOTED_IDENTIFIER`
  - The `NUMERIC_ROUNDABORT` session option is set to OFF.
- The Query Optimizer finds a match between the view index columns and elements in the query, such as the following:
  - Search condition predicates in the WHERE clause
  - Join operations
  - Aggregate functions
  - `GROUP BY` clauses
  - Table references
- The estimated cost for using the index has the lowest cost of any access mechanisms considered by the Query Optimizer.
- Every table referenced in the query (either directly, or by expanding a view to access its underlying tables) that corresponds to a table reference in the indexed view must have the same set of hints applied on it in the query.

> **NOTE**
>
> The `READCOMMITTED` and `READCOMMITTEDLOCK` hints are always considered different hints in this context, regardless of the current transaction isolation level.

Other than the requirements for the `SET` options and table hints, these are the same rules that the Query Optimizer uses to determine whether a table index covers a query. Nothing else has to be specified in the query for an indexed view to be used.

A query does not have to explicitly reference an indexed view in the `FROM` clause for the Query Optimizer to use the indexed view. If the query contains references to columns in the base tables that are also present in the indexed view, and the Query Optimizer estimates that using the indexed view provides the lowest cost access mechanism, the Query Optimizer chooses the indexed view, similar to the way it chooses base table indexes when they are not directly referenced in a query. The Query Optimizer may choose the view when it contains columns that are not referenced by the query, as long as the view offers the lowest cost option for covering one or more of the columns specified in the query.

The Query Optimizer treats an indexed view referenced in the `FROM` clause as a standard view. The Query Optimizer expands the definition of the view into the query at the start of the optimization process. Then, indexed view matching is performed. The indexed view may be used in the final execution plan selected by the Query Optimizer, or instead, the plan may materialize necessary data from the view by accessing the base tables referenced by the view. The Query Optimizer chooses the lowest-cost alternative.

#### Using Hints with Indexed Views

You can prevent view indexes from being used for a query by using the `EXPAND VIEWS` query hint, or you can use the `NOEXPAND` table hint to force the use of an index for an indexed view specified in the `FROM` clause of a query. However, you should let the Query Optimizer dynamically determine the best access methods to use for each query. Limit your use of `EXPAND` and `NOEXPAND` to specific cases where testing has shown that they improve

performance significantly.

The `EXPAND VIEWS` option specifies that the Query Optimizer not use any view indexes for the whole query.

When `NOEXPAND` is specified for a view, the Query Optimizer considers using any indexes defined on the view. `NOEXPAND` specified with the optional `INDEX()` clause forces the Query Optimizer to use the specified indexes. `NOEXPAND` can be specified only for an indexed view and cannot be specified for a view not indexed.

When neither `NOEXPAND` nor `EXPAND VIEWS` is specified in a query that contains a view, the view is expanded to access underlying tables. If the query that makes up the view contains any table hints, these hints are propagated to the underlying tables. (This process is explained in more detail in View Resolution.) As long as the set of hints that exists on the underlying tables of the view are identical to each other, the query is eligible to be matched with an indexed view. Most of the time, these hints will match each other, because they are being inherited directly from the view. However, if the query references tables instead of views, and the hints applied directly on these tables are not identical, then such a query is not eligible for matching with an indexed view. If the `INDEX`, `PAGLOCK`, `ROWLOCK`, `TABLOCKX`, `UPDLOCK`, or `XLOCK` hints apply to the tables referenced in the query after view expansion, the query is not eligible for indexed view matching.

If a table hint in the form of `INDEX (index_val[ ,...n] )` references a view in a query and you do not also specify the `NOEXPAND` hint, the index hint is ignored. To specify use of a particular index, use `NOEXPAND`.

Generally, when the Query Optimizer matches an indexed view to a query, any hints specified on the tables or views in the query are applied directly to the indexed view. If the Query Optimizer chooses not to use an indexed view, any hints are propagated directly to the tables referenced in the view. For more information, see View Resolution. This propagation does not apply to join hints. They are applied only in their original position in the query. Join hints are not considered by the Query Optimizer when matching queries to indexed views. If a query plan uses an indexed view that matches part of a query that contains a join hint, the join hint is not used in the plan.

Hints are not allowed in the definitions of indexed views. In compatibility mode 80 and higher, SQL Server ignores hints inside indexed view definitions when maintaining them, or when executing queries that use indexed views. Although using hints in indexed view definitions will not produce a syntax error in 80 compatibility mode, they are ignored.

**Resolving Distributed Partitioned Views**

The SQL Server query processor optimizes the performance of distributed partitioned views. The most important aspect of distributed partitioned view performance is minimizing the amount of data transferred between member servers.

SQL Server builds intelligent, dynamic plans that make efficient use of distributed queries to access data from remote member tables:

- The Query Processor first uses OLE DB to retrieve the check constraint definitions from each member table. This allows the query processor to map the distribution of key values across the member tables.
- The Query Processor compares the key ranges specified in an SQL statement `WHERE` clause to the map that shows how the rows are distributed in the member tables. The query processor then builds a query execution plan that uses distributed queries to retrieve only those remote rows that are required to complete the SQL statement. The execution plan is also built in such a way that any access to remote member tables, for either data or metadata, are delayed until the information is required.

For example, consider a system where a customers table is partitioned across Server1 (`CustomerID` from 1 through 3299999), Server2 (`CustomerID` from 3300000 through 6599999), and Server3 (`CustomerID` from 6600000 through 9999999).

Consider the execution plan built for this query executed on Server1:

```
SELECT *
FROM CompanyData.dbo.Customers
WHERE CustomerID BETWEEN 3200000 AND 3400000;
```

The execution plan for this query extracts the rows with `CustomerID` key values from 3200000 through 3299999 from the local member table, and issues a distributed query to retrieve the rows with key values from 3300000 through 3400000 from Server2.

The SQL Server Query Processor can also build dynamic logic into query execution plans for SQL statements in which the key values are not known when the plan must be built. For example, consider this stored procedure:

```
CREATE PROCEDURE GetCustomer @CustomerIDParameter INT
AS
SELECT *
FROM CompanyData.dbo.Customers
WHERE CustomerID = @CustomerIDParameter;
```

SQL Server cannot predict what key value will be supplied by the `@CustomerIDParameter` parameter every time the procedure is executed. Because the key value cannot be predicted, the query processor also cannot predict which member table will have to be accessed. To handle this case, SQL Server builds an execution plan that has conditional logic, referred to as dynamic filters, to control which member table is accessed, based on the input parameter value. Assuming the `GetCustomer` stored procedure was executed on Server1, the execution plan logic can be represented as shown in the following:

```
IF @CustomerIDParameter BETWEEN 1 and 3299999
   Retrieve row from local table CustomerData.dbo.Customer_33
ELSE IF @CustomerIDParameter BETWEEN 3300000 and 6599999
   Retrieve row from linked table Server2.CustomerData.dbo.Customer_66
ELSE IF @CustomerIDParameter BETWEEN 6600000 and 9999999
   Retrieve row from linked table Server3.CustomerData.dbo.Customer_99
```

SQL Server sometimes builds these types of dynamic execution plans even for queries that are not parameterized. The Query Optimizer may parameterize a query so that the execution plan can be reused. If the Query Optimizer parameterizes a query referencing a partitioned view, the Query Optimizer can no longer assume the required rows will come from a specified base table. It will then have to use dynamic filters in the execution plan.

## Stored Procedure and Trigger Execution

SQL Server stores only the source for stored procedures and triggers. When a stored procedure or trigger is first executed, the source is compiled into an execution plan. If the stored procedure or trigger is again executed before the execution plan is aged from memory, the relational engine detects the existing plan and reuses it. If the plan has aged out of memory, a new plan is built. This process is similar to the process SQL Server follows for all SQL statements. The main performance advantage that stored procedures and triggers have in SQL Server compared with batches of dynamic SQL is that their SQL statements are always the same. Therefore, the relational engine easily matches them with any existing execution plans. Stored procedure and trigger plans are easily reused.

The execution plan for stored procedures and triggers is executed separately from the execution plan for the batch calling the stored procedure or firing the trigger. This allows for greater reuse of the stored procedure and trigger execution plans.

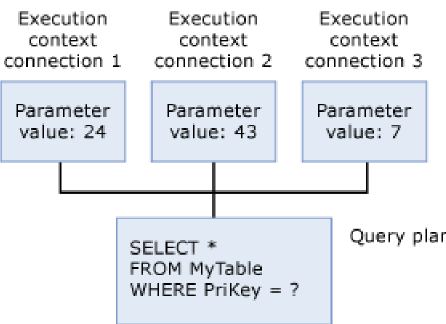
## Execution Plan Caching and Reuse

SQL Server has a pool of memory that is used to store both execution plans and data buffers. The percentage of the pool allocated to either execution plans or data buffers fluctuates dynamically, depending on the state of the

system. The part of the memory pool that is used to store execution plans is referred to as the plan cache.

SQL Server execution plans have the following main components:

- Query Execution Plan The bulk of the execution plan is a re-entrant, read-only data structure used by any number of users. This is referred to as the query plan. No user context is stored in the query plan. There are never more than one or two copies of the query plan in memory: one copy for all serial executions and another for all parallel executions. The parallel copy covers all parallel executions, regardless of their degree of parallelism.
- Execution Context Each user that is currently executing the query has a data structure that holds the data specific to their execution, such as parameter values. This data structure is referred to as the execution context. The execution context data structures are reused. If a user executes a query and one of the structures is not being used, it is reinitialized with the context for the new user.




When any SQL statement is executed in SQL Server, the relational engine first looks through the plan cache to verify that an existing execution plan for the same SQL statement exists. SQL Server reuses any existing plan it finds, saving the overhead of recompiling the SQL statement. If no existing execution plan exists, SQL Server generates a new execution plan for the query.

SQL Server has an efficient algorithm to find any existing execution plans for any specific SQL statement. In most systems, the minimal resources that are used by this scan are less than the resources that are saved by being able to reuse existing plans instead of compiling every SQL statement.

The algorithms to match new SQL statements to existing, unused execution plans in the cache require that all object references be fully qualified. For example, the first of these `SELECT` statements is not matched with an existing plan, and the second is matched:

```
SELECT * FROM Person;

SELECT * FROM Person.Person;
```

**Removing Execution Plans from the Plan Cache**

Execution plans remain in the plan cache as long as there is enough memory to store them. When memory pressure exists, the SQL Server Database Engine uses a cost-based approach to determine which execution plans to remove from the plan cache. To make a cost-based decision, the SQL Server Database Engine increases and decreases a current cost variable for each execution plan according to the following factors.

When a user process inserts an execution plan into the cache, the user process sets the current cost equal to the original query compile cost; for ad-hoc execution plans, the user process sets the current cost to zero. Thereafter, each time a user process references an execution plan, it resets the current cost to the original compile cost; for ad-hoc execution plans the user process increases the current cost. For all plans, the maximum value for the current cost is the original compile cost.

When memory pressure exists, the SQL Server Database Engine responds by removing execution plans from the plan cache. To determine which plans to remove, the SQL Server Database Engine repeatedly examines the state of each execution plan and removes plans when their current cost is zero. An execution plan with zero current cost

is not removed automatically when memory pressure exists; it is removed only when the SQL Server Database Engine examines the plan and the current cost is zero. When examining an execution plan, the SQL Server Database Engine pushes the current cost towards zero by decreasing the current cost if a query is not currently using the plan.

The SQL Server Database Engine repeatedly examines the execution plans until enough have been removed to satisfy memory requirements. While memory pressure exists, an execution plan may have its cost increased and decreased more than once. When memory pressure no longer exists, the SQL Server Database Engine stops decreasing the current cost of unused execution plans and all execution plans remain in the plan cache, even if their cost is zero.

The SQL Server Database Engine uses the resource monitor and user worker threads to free memory from the plan cache in response to memory pressure. The resource monitor and user worker threads can examine plans run concurrently to decrease the current cost for each unused execution plan. The resource monitor removes execution plans from the plan cache when global memory pressure exists. It frees memory to enforce policies for system memory, process memory, resource pool memory, and maximum size for all caches.

The maximum size for all caches is a function of the buffer pool size and cannot exceed the maximum server memory. For more information on configuring the maximum server memory, see the `max server memory` setting in `sp_configure`.

The user worker threads remove execution plans from the plan cache when single cache memory pressure exists. They enforce policies for maximum single cache size and maximum single cache entries.

The following examples illustrate which execution plans get removed from the plan cache:

- An execution plan is frequently referenced so that its cost never goes to zero. The plan remains in the plan cache and is not removed unless there is memory pressure and the current cost is zero.
- An ad-hoc execution plan is inserted and is not referenced again before memory pressure exists. Since ad-hoc plans are initialized with a current cost of zero, when the SQL Server Database Engine examines the execution plan, it will see the zero current cost and remove the plan from the plan cache. The ad-hoc execution plan remains in the plan cache with a zero current cost when memory pressure does not exist.

To manually remove a single plan or all plans from the cache, use DBCC FREEPROCCACHE.

**Recompiling Execution Plans**

Certain changes in a database can cause an execution plan to be either inefficient or invalid, based on the new state of the database. SQL Server detects the changes that invalidate an execution plan and marks the plan as not valid. A new plan must then be recompiled for the next connection that executes the query. The conditions that invalidate a plan include the following:

- Changes made to a table or view referenced by the query (`ALTER TABLE` and `ALTER VIEW`).
- Changes made to a single procedure, which would drop all plans for that procedure from the cache (`ALTER PROCEDURE`).
- Changes to any indexes used by the execution plan.
- Updates on statistics used by the execution plan, generated either explicitly from a statement, such as `UPDATE STATISTICS`, or generated automatically.
- Dropping an index used by the execution plan.
- An explicit call to `sp_recompile`.
- Large numbers of changes to keys (generated by `INSERT` or `DELETE` statements from other users that modify a table referenced by the query).
- For tables with triggers, if the number of rows in the inserted or deleted tables grows significantly.
- Executing a stored procedure using the `WITH RECOMPILE` option.

Most recompilations are required either for statement correctness or to obtain potentially faster query execution

plans.

In SQL Server 2000, whenever a statement within a batch causes recompilation, the whole batch, whether submitted through a stored procedure, trigger, ad-hoc batch, or prepared statement, is recompiled. Starting with SQL Server 2005, only the statement inside the batch that causes recompilation is recompiled. Because of this difference, recompilation counts in SQL Server 2000 and later releases are not comparable. Also, there are more types of recompilations in SQL Server 2005 and later because of its expanded feature set.

Statement-level recompilation benefits performance because, in most cases, a small number of statements causes recompilations and their associated penalties, in terms of CPU time and locks. These penalties are therefore avoided for the other statements in the batch that do not have to be recompiled.

The `sql_statement_recompile` extended event (xEvent) reports statement-level recompilations. This xEvent occurs when a statement-level recompilation is required by any kind of batch. This includes stored procedures, triggers, ad hoc batches and queries. Batches may be submitted through several interfaces, including sp_executesql, dynamic SQL, Prepare methods or Execute methods. The `recompile_cause` column of `sql_statement_recompile` xEvent contains an integer code that indicates the reason for the recompilation. The following table contains the possible reasons:

| | |
|---|---|
| Schema changed | Statistics changed |
| Deferred compile | SET option changed |
| Temporary table changed | Remote rowset changed |
| `FOR BROWSE` permission changed | Query notification environment changed |
| Partitioned view changed | Cursor options changed |
| `OPTION (RECOMPILE)` requested | Parameterized plan flushed |
| Plan affecting database version changed | Query Store plan forcing policy changed |
| Query Store plan forcing failed | Query Store missing the plan |

**NOTE**

In SQL Server versions where xEvents are not available, then the SQL Server Profiler SP:Recompile trace event can be used for the same purpose of reporting statement-level recompilations. The trace event SQL:StmtRecompile also reports statement-level recompilations, and this trace event can also be used to track and debug recompilations. Whereas SP:Recompile generates only for stored procedures and triggers, SQL:StmtRecompile generates for stored procedures, triggers, ad-hoc batches, batches that are executed by using `sp_executesql`, prepared queries, and dynamic SQL. The *EventSubClass* column of SP:Recompile and SQL:StmtRecompile contains an integer code that indicates the reason for the recompilation. The codes are described here.

**NOTE**

When the `AUTO_UPDATE_STATISTICS` database option is set to `ON`, queries are recompiled when they target tables or indexed views whose statistics have been updated or whose cardinalities have changed significantly since the last execution. This behavior applies to standard user-defined tables, temporary tables, and the inserted and deleted tables created by DML triggers. If query performance is affected by excessive recompilations, consider changing this setting to `OFF`. When the `AUTO_UPDATE_STATISTICS` database option is set to `OFF`, no recompilations occur based on statistics or cardinality changes, with the exception of the inserted and deleted tables that are created by DML `INSTEAD OF` triggers. Because these tables are created in tempdb, the recompilation of queries that access them depends on the setting of `AUTO_UPDATE_STATISTICS` in tempdb. Note that in SQL Server 2000, queries continue to recompile based on cardinality changes to the DML trigger inserted and deleted tables, even when this setting is `OFF`.

### Parameters and Execution Plan Reuse

The use of parameters, including parameter markers in ADO, OLE DB, and ODBC applications, can increase the reuse of execution plans.

**WARNING**

Using parameters or parameter markers to hold values that are typed by end users is more secure than concatenating the values into a string that is then executed by using either a data access API method, the `EXECUTE` statement, or the `sp_executesql` stored procedure.

The only difference between the following two `SELECT` statements is the values that are compared in the `WHERE` clause:

```
SELECT *
FROM AdventureWorks2014.Production.Product
WHERE ProductSubcategoryID = 1;
```

```
SELECT *
FROM AdventureWorks2014.Production.Product
WHERE ProductSubcategoryID = 4;
```

The only difference between the execution plans for these queries is the value stored for the comparison against the `ProductSubcategoryID` column. While the goal is for SQL Server to always recognize that the statements generate essentially the same plan and reuse the plans, SQL Server sometimes does not detect this in complex SQL statements.

Separating constants from the SQL statement by using parameters helps the relational engine recognize duplicate plans. You can use parameters in the following ways:

- In Transact-SQL, use `sp_executesql`:

  ```
  DECLARE @MyIntParm INT
  SET @MyIntParm = 1
  EXEC sp_executesql
    N'SELECT *
    FROM AdventureWorks2014.Production.Product
    WHERE ProductSubcategoryID = @Parm',
    N'@Parm INT',
    @MyIntParm
  ```

  This method is recommended for Transact-SQL scripts, stored procedures, or triggers that generate SQL

statements dynamically.

- ADO, OLE DB, and ODBC use parameter markers. Parameter markers are question marks (?) that replace a constant in an SQL statement and are bound to a program variable. For example, you would do the following in an ODBC application:
  - Use `SQLBindParameter` to bind an integer variable to the first parameter marker in an SQL statement.
  - Put the integer value in the variable.
  - Execute the statement, specifying the parameter marker (?):

    ```
    SQLExecDirect(hstmt,
    "SELECT *
    FROM AdventureWorks2014.Production.Product
    WHERE ProductSubcategoryID = ?",
    SQL_NTS);
    ```

    The SQL Server Native Client OLE DB Provider and the SQL Server Native Client ODBC driver included with SQL Server use `sp_executesql` to send statements to SQL Server when parameter markers are used in applications.

- To design stored procedures, which use parameters by design.

If you do not explicitly build parameters into the design of your applications, you can also rely on the SQL Server Query Optimizer to automatically parameterize certain queries by using the default behavior of simple parameterization. Alternatively, you can force the Query Optimizer to consider parameterizing all queries in the database by setting the `PARAMETERIZATION` option of the `ALTER DATABASE` statement to `FORCED`.

When forced parameterization is enabled, simple parameterization can still occur. For example, the following query cannot be parameterized according to the rules of forced parameterization:

```
SELECT * FROM Person.Address
WHERE AddressID = 1 + 2;
```

However, it can be parameterized according to simple parameterization rules. When forced parameterization is tried but fails, simple parameterization is still subsequently tried.

**Simple Parameterization**

In SQL Server, using parameters or parameter markers in Transact-SQL statements increases the ability of the relational engine to match new SQL statements with existing, previously-compiled execution plans.

> **WARNING**
>
> Using parameters or parameter markers to hold values typed by end users is more secure than concatenating the values into a string that is then executed using either a data access API method, the `EXECUTE` statement, or the `sp_executesql` stored procedure.

If a SQL statement is executed without parameters, SQL Server parameterizes the statement internally to increase the possibility of matching it against an existing execution plan. This process is called simple parameterization. In SQL Server 2000, the process was referred to as auto-parameterization.

Consider this statement:

```
SELECT * FROM AdventureWorks2014.Production.Product
WHERE ProductSubcategoryID = 1;
```

The value 1 at the end of the statement can be specified as a parameter. The relational engine builds the execution plan for this batch as if a parameter had been specified in place of the value 1. Because of this simple parameterization, SQL Server recognizes that the following two statements generate essentially the same execution plan and reuses the first plan for the second statement:

```
SELECT * FROM AdventureWorks2014.Production.Product
WHERE ProductSubcategoryID = 1;
```

```
SELECT * FROM AdventureWorks2014.Production.Product
WHERE ProductSubcategoryID = 4;
```

When processing complex SQL statements, the relational engine may have difficulty determining which expressions can be parameterized. To increase the ability of the relational engine to match complex SQL statements to existing, unused execution plans, explicitly specify the parameters using either sp_executesql or parameter markers.

> **NOTE**
>
> When the +, -, *, /, or % arithmetic operators are used to perform implicit or explicit conversion of int, smallint, tinyint, or bigint constant values to the float, real, decimal or numeric data types, SQL Server applies specific rules to calculate the type and precision of the expression results. However, these rules differ, depending on whether the query is parameterized or not. Therefore, similar expressions in queries can, in some cases, produce differing results.

Under the default behavior of simple parameterization, SQL Server parameterizes a relatively small class of queries. However, you can specify that all queries in a database be parameterized, subject to certain limitations, by setting the `PARAMETERIZATION` option of the `ALTER DATABASE` command to `FORCED`. Doing so may improve the performance of databases that experience high volumes of concurrent queries by reducing the frequency of query compilations.

Alternatively, you can specify that a single query, and any others that are syntactically equivalent but differ only in their parameter values, be parameterized.

#### Forced Parameterization

You can override the default simple parameterization behavior of SQL Server by specifying that all `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements in a database be parameterized, subject to certain limitations. Forced parameterization is enabled by setting the `PARAMETERIZATION` option to `FORCED` in the `ALTER DATABASE` statement. Forced parameterization may improve the performance of certain databases by reducing the frequency of query compilations and recompilations. Databases that may benefit from forced parameterization are generally those that experience high volumes of concurrent queries from sources such as point-of-sale applications.

When the `PARAMETERIZATION` option is set to `FORCED`, any literal value that appears in a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement, submitted in any form, is converted to a parameter during query compilation. The exceptions are literals that appear in the following query constructs:

- `INSERT...EXECUTE` statements.
- Statements inside the bodies of stored procedures, triggers, or user-defined functions. SQL Server already reuses query plans for these routines.
- Prepared statements that have already been parameterized on the client-side application.
- Statements that contain XQuery method calls, where the method appears in a context where its arguments would typically be parameterized, such as a `WHERE` clause. If the method appears in a context where its arguments would not be parameterized, the rest of the statement is parameterized.
- Statements inside a Transact-SQL cursor. (`SELECT` statements inside API cursors are parameterized.)

- Deprecated query constructs.
- Any statement that is run in the context of `ANSI_PADDING` or `ANSI_NULLS` set to `OFF`.
- Statements that contain more than 2,097 literals that are eligible for parameterization.
- Statements that reference variables, such as `WHERE T.col2 >= @bb`.
- Statements that contain the `RECOMPILE` query hint.
- Statements that contain a `COMPUTE` clause.
- Statements that contain a `WHERE CURRENT OF` clause.

Additionally, the following query clauses are not parameterized. Note that in these cases, only the clauses are not parameterized. Other clauses within the same query may be eligible for forced parameterization.

- The <select_list> of any `SELECT` statement. This includes `SELECT` lists of subqueries and `SELECT` lists inside `INSERT` statements.
- Subquery `SELECT` statements that appear inside an `IF` statement.
- The `TOP`, `TABLESAMPLE`, `HAVING`, `GROUP BY`, `ORDER BY`, `OUTPUT...INTO`, or `FOR XM` L clauses of a query.
- Arguments, either direct or as subexpressions, to `OPENROWSET`, `OPENQUERY`, `OPENDATASOURCE`, `OPENXML`, or any `FULLTEXT` operator.
- The pattern and escape_character arguments of a `LIKE` clause.
- The style argument of a `CONVERT` clause.
- Integer constants inside an `IDENTITY` clause.
- Constants specified by using ODBC extension syntax.
- Constant-foldable expressions that are arguments of the +, -, *, /, and % operators. When considering eligibility for forced parameterization, SQL Server considers an expression to be constant-foldable when either of the following conditions is true:
  - No columns, variables, or subqueries appear in the expression.
  - The expression contains a `CASE` clause.
- Arguments to query hint clauses. These include the `number_of_rows` argument of the `FAST` query hint, the `number_of_processors` argument of the `MAXDOP` query hint, and the number argument of the `MAXRECURSION` query hint.

Parameterization occurs at the level of individual Transact-SQL statements. In other words, individual statements in a batch are parameterized. After compiling, a parameterized query is executed in the context of the batch in which it was originally submitted. If an execution plan for a query is cached, you can determine whether the query was parameterized by referencing the sql column of the sys.syscacheobjects dynamic management view. If a query is parameterized, the names and data types of parameters come before the text of the submitted batch in this column, such as (@1 tinyint).

> **NOTE**
>
> Parameter names are arbitrary. Users or applications should not rely on a particular naming order. Also, the following can change between versions of SQL Server and Service Pack upgrades: Parameter names, the choice of literals that are parameterized, and the spacing in the parameterized text.

**Data Types of Parameters**

When SQL Server parameterizes literals, the parameters are converted to the following data types:

- Integer literals whose size would otherwise fit within the int data type parameterize to int. Larger integer literals that are parts of predicates that involve any comparison operator (includes <, <=, =, !=, >, >=, , !<, !>, <>, `ALL`, `ANY`, `SOME`, `BETWEEN`, and `IN`) parameterize to numeric(38,0). Larger literals that are not parts of predicates that involve comparison operators parameterize to numeric whose precision is just large enough to support its size and whose scale is 0.

- Fixed-point numeric literals that are parts of predicates that involve comparison operators parameterize to numeric whose precision is 38 and whose scale is just large enough to support its size. Fixed-point numeric literals that are not parts of predicates that involve comparison operators parameterize to numeric whose precision and scale are just large enough to support its size.
- Floating point numeric literals parameterize to float(53).
- Non-Unicode string literals parameterize to varchar(8000) if the literal fits within 8,000 characters, and to varchar(max) if it is larger than 8,000 characters.
- Unicode string literals parameterize to nvarchar(4000) if the literal fits within 4,000 Unicode characters, and to nvarchar(max) if the literal is larger than 4,000 characters.
- Binary literals parameterize to varbinary(8000) if the literal fits within 8,000 bytes. If it is larger than 8,000 bytes, it is converted to varbinary(max).
- Money type literals parameterize to money.

#### Guidelines for Using Forced Parameterization

Consider the following when you set the `PARAMETERIZATION` option to FORCED:

- Forced parameterization, in effect, changes the literal constants in a query to parameters when compiling a query. Therefore, the Query Optimizer might choose suboptimal plans for queries. In particular, the Query Optimizer is less likely to match the query to an indexed view or an index on a computed column. It may also choose suboptimal plans for queries posed on partitioned tables and distributed partitioned views. Forced parameterization should not be used for environments that rely heavily on indexed views and indexes on computed columns. Generally, the `PARAMETERIZATION FORCED` option should only be used by experienced database administrators after determining that doing this does not adversely affect performance.
- Distributed queries that reference more than one database are eligible for forced parameterization as long as the `PARAMETERIZATION` option is set to `FORCED` in the database whose context the query is running.
- Setting the `PARAMETERIZATION` option to `FORCED` flushes all query plans from the plan cache of a database, except those that currently are compiling, recompiling, or running. Plans for queries that are compiling or running during the setting change are parameterized the next time the query is executed.
- Setting the `PARAMETERIZATION` option is an online operation that it requires no database-level exclusive locks.
- The current setting of the `PARAMETERIZATION` option is preserved when reattaching or restoring a database.

You can override the behavior of forced parameterization by specifying that simple parameterization be attempted on a single query, and any others that are syntactically equivalent but differ only in their parameter values. Conversely, you can specify that forced parameterization be attempted on only a set of syntactically equivalent queries, even if forced parameterization is disabled in the database. Plan guides are used for this purpose.

> **NOTE**
>
> When the `PARAMETERIZATION` option is set to `FORCED`, the reporting of error messages may differ from when the `PARAMETERIZATION` option is set to `SIMPLE`: multiple error messages may be reported under forced parameterization, where fewer messages would be reported under simple parameterization, and the line numbers in which errors occur may be reported incorrectly.

### Preparing SQL Statements

The SQL Server relational engine introduces full support for preparing SQL statements before they are executed. If an application has to execute an SQL statement several times, it can use the database API to do the following:

- Prepare the statement once. This compiles the SQL statement into an execution plan.
- Execute the precompiled execution plan every time it has to execute the statement. This prevents having to recompile the SQL statement on each execution after the first time.
  Preparing and executing statements is controlled by API functions and methods. It is not part of the Transact-SQL language. The prepare/execute model of executing SQL statements is supported by the SQL Server

Native Client OLE DB Provider and the SQL Server Native Client ODBC driver. On a prepare request, either the provider or the driver sends the statement to SQL Server with a request to prepare the statement. SQL Server compiles an execution plan and returns a handle for that plan to the provider or driver. On an execute request, either the provider or the driver sends the server a request to execute the plan that is associated with the handle.

Prepared statements cannot be used to create temporary objects on SQL Server. Prepared statements cannot reference system stored procedures that create temporary objects, such as temporary tables. These procedures must be executed directly.

Excess use of the prepare/execute model can degrade performance. If a statement is executed only once, a direct execution requires only one network round-trip to the server. Preparing and executing an SQL statement executed only one time requires an extra network round-trip; one trip to prepare the statement and one trip to execute it.

Preparing a statement is more effective if parameter markers are used. For example, assume that an application is occasionally asked to retrieve product information from the `AdventureWorks` sample database. There are two ways the application can do this.

Using the first way, the application can execute a separate query for each product requested:

```
SELECT * FROM AdventureWorks2014.Production.Product
WHERE ProductID = 63;
```

Using the second way, the application does the following:

1. Prepares a statement that contains a parameter marker (?):
   `sql SELECT * FROM AdventureWorks2014.Production.Product WHERE ProductID = ?;`
2. Binds a program variable to the parameter marker.
3. Each time product information is needed, fills the bound variable with the key value and executes the statement.

The second way is more efficient when the statement is executed more than three times.

In SQL Server, the prepare/execute model has no significant performance advantage over direct execution, because of the way SQL Server reuses execution plans. SQL Server has efficient algorithms for matching current SQL statements with execution plans that are generated for prior executions of the same SQL statement. If an application executes a SQL statement with parameter markers multiple times, SQL Server will reuse the execution plan from the first execution for the second and subsequent executions (unless the plan ages from the plan cache). The prepare/execute model still has these benefits:

- Finding an execution plan by an identifying handle is more efficient than the algorithms used to match an SQL statement to existing execution plans.
- The application can control when the execution plan is created and when it is reused.
- The prepare/execute model is portable to other databases, including earlier versions of SQL Server.

**Parameter Sniffing**

"Parameter sniffing" refers to a process whereby SQL Server "sniffs" the current parameter values during compilation or recompilation, and passes it along to the Query Optimizer so that they can be used to generate potentially more efficient query execution plans.

Parameter values are sniffed during compilation or recompilation for the following types of batches:

- Stored procedures
- Queries submitted via sp_executesql
- Prepared queries

> **NOTE**
>
> For queries using the `RECOMPILE` hint, both parameter values and current values of local variables are sniffed. The values sniffed (of parameters and local variables) are those that exist at the place in the batch just before the statement with the `RECOMPILE` hint. In particular, for parameters, the values that came along with the batch invocation call are not sniffed.

## Parallel Query Processing

SQL Server provides parallel queries to optimize query execution and index operations for computers that have more than one microprocessor (CPU). Because SQL Server can perform a query or index operation in parallel by using several operating system worker threads, the operation can be completed quickly and efficiently.

During query optimization, SQL Server looks for queries or index operations that might benefit from parallel execution. For these queries, SQL Server inserts exchange operators into the query execution plan to prepare the query for parallel execution. An exchange operator is an operator in a query execution plan that provides process management, data redistribution, and flow control. The exchange operator includes the `Distribute Streams`, `Repartition Streams`, and `Gather Streams` logical operators as subtypes, one or more of which can appear in the Showplan output of a query plan for a parallel query.

After exchange operators are inserted, the result is a parallel-query execution plan. A parallel-query execution plan can use more than one worker thread. A serial execution plan, used by a nonparallel query, uses only one worker thread for its execution. The actual number of worker threads used by a parallel query is determined at query plan execution initialization and is determined by the complexity of the plan and the degree of parallelism. Degree of parallelism determines the maximum number of CPUs that are being used; it does not mean the number of worker threads that are being used. The degree of parallelism value is set at the server level and can be modified by using the sp_configure system stored procedure. You can override this value for individual query or index statements by specifying the `MAXDOP` query hint or `MAXDOP` index option.

The SQL Server Query Optimizer does not use a parallel execution plan for a query if any one of the following conditions is true:

- The serial execution cost of the query is not high enough to consider an alternative, parallel execution plan.
- A serial execution plan is considered faster than any possible parallel execution plan for the particular query.
- The query contains scalar or relational operators that cannot be run in parallel. Certain operators can cause a section of the query plan to run in serial mode, or the whole plan to run in serial mode.

### Degree of Parallelism

SQL Server automatically detects the best degree of parallelism for each instance of a parallel query execution or index data definition language (DDL) operation. It does this based on the following criteria:

1. Whether SQL Server is running on a computer that has more than one microprocessor or CPU, such as a symmetric multiprocessing computer (SMP).
   Only computers that have more than one CPU can use parallel queries.

2. Whether sufficient worker threads are available.
   Each query or index operation requires a certain number of worker threads to execute. Executing a parallel plan requires more worker threads than a serial plan, and the number of required worker threads increases with the degree of parallelism. When the worker thread requirement of the parallel plan for a specific degree of parallelism cannot be satisfied, the SQL Server Database Engine decreases the degree of parallelism automatically or completely abandons the parallel plan in the specified workload context. It then executes the serial plan (one worker thread).

3. The type of query or index operation executed.
   Index operations that create or rebuild an index, or drop a clustered index and queries that use CPU cycles

heavily are the best candidates for a parallel plan. For example, joins of large tables, large aggregations, and sorting of large result sets are good candidates. Simple queries, frequently found in transaction processing applications, find the additional coordination required to execute a query in parallel outweigh the potential performance boost. To distinguish between queries that benefit from parallelism and those that do not benefit, the SQL Server Database Engine compares the estimated cost of executing the query or index operation with the cost threshold for parallelism value. Users can change the default value of 5 using sp_configure if proper testing found that a different value is better suited for the running workload.

4. Whether there are a sufficient number of rows to process.
   If the Query Optimizer determines that the number of rows is too low, it does not introduce exchange operators to distribute the rows. Consequently, the operators are executed serially. Executing the operators in a serial plan avoids scenarios when the startup, distribution, and coordination costs exceed the gains achieved by parallel operator execution.

5. Whether current distribution statistics are available.
   If the highest degree of parallelism is not possible, lower degrees are considered before the parallel plan is abandoned.
   For example, when you create a clustered index on a view, distribution statistics cannot be evaluated, because the clustered index does not yet exist. In this case, the SQL Server Database Engine cannot provide the highest degree of parallelism for the index operation. However, some operators, such as sorting and scanning, can still benefit from parallel execution.

> **NOTE**
>
> Parallel index operations are only available in SQL Server Enterprise, Developer, and Evaluation editions.

At execution time, the SQL Server Database Engine determines whether the current system workload and configuration information previously described allow for parallel execution. If parallel execution is warranted, the SQL Server Database Engine determines the optimal number of worker threads and spreads the execution of the parallel plan across those worker threads. When a query or index operation starts executing on multiple worker threads for parallel execution, the same number of worker threads is used until the operation is completed. The SQL Server Database Engine re-examines the optimal number of worker thread decisions every time an execution plan is retrieved from the plan cache. For example, one execution of a query can result in the use of a serial plan, a later execution of the same query can result in a parallel plan using three worker threads, and a third execution can result in a parallel plan using four worker threads.

In a parallel query execution plan, the insert, update, and delete operators are executed serially. However, the WHERE clause of an UPDATE or a DELETE statement, or the SELECT part of an INSERT statement may be executed in parallel. The actual data changes are then serially applied to the database.

Static and keyset-driven cursors can be populated by parallel execution plans. However, the behavior of dynamic cursors can be provided only by serial execution. The Query Optimizer always generates a serial execution plan for a query that is part of a dynamic cursor.

#### Overriding Degrees of Parallelism

You can use the max degree of parallelism (MAXDOP) server configuration option (ALTER DATABASE SCOPED CONFIGURATION on SQL Database ) to limit the number of processors to use in parallel plan execution. The max degree of parallelism option can be overridden for individual query and index operation statements by specifying the MAXDOP query hint or MAXDOP index option. MAXDOP provides more control over individual queries and index operations. For example, you can use the MAXDOP option to control, by increasing or reducing, the number of processors dedicated to an online index operation. In this way, you can balance the resources used by an index operation with those of the concurrent users.

Setting the max degree of parallelism option to 0 (default) enables SQL Server to use all available processors up to a maximum of 64 processors in a parallel plan execution. Although SQL Server sets a runtime target of 64

logical processors when MAXDOP option is set to 0, a different value can be manually set if needed. Setting MAXDOP to 0 for queries and indexes allows SQL Server to use all available processors up to a maximum of 64 processors for the given queries or indexes in a parallel plan execution. MAXDOP is not an enforced value for all parallel queries, but rather a tentative target for all queries eligible for parallelism. This means that if not enough worker threads are available at runtime, a query may execute with a lower degree of parallelism than the MAXDOP server configuration option.

Refer to this Microsoft Support Article for best practices on configuring MAXDOP.

**Parallel Query Example**

The following query counts the number of orders placed in a specific quarter, starting on April 1, 2000, and in which at least one line item of the order was received by the customer later than the committed date. This query lists the count of such orders grouped by each order priority and sorted in ascending priority order.

This example uses theoretical table and column names.

```
SELECT o_orderpriority, COUNT(*) AS Order_Count
FROM orders
WHERE o_orderdate >= '2000/04/01'
   AND o_orderdate < DATEADD (mm, 3, '2000/04/01')
   AND EXISTS
         (
          SELECT *
            FROM    lineitem
            WHERE l_orderkey = o_orderkey
               AND l_commitdate < l_receiptdate
         )
   GROUP BY o_orderpriority
   ORDER BY o_orderpriority
```

Assume the following indexes are defined on the `lineitem` and `orders` tables:

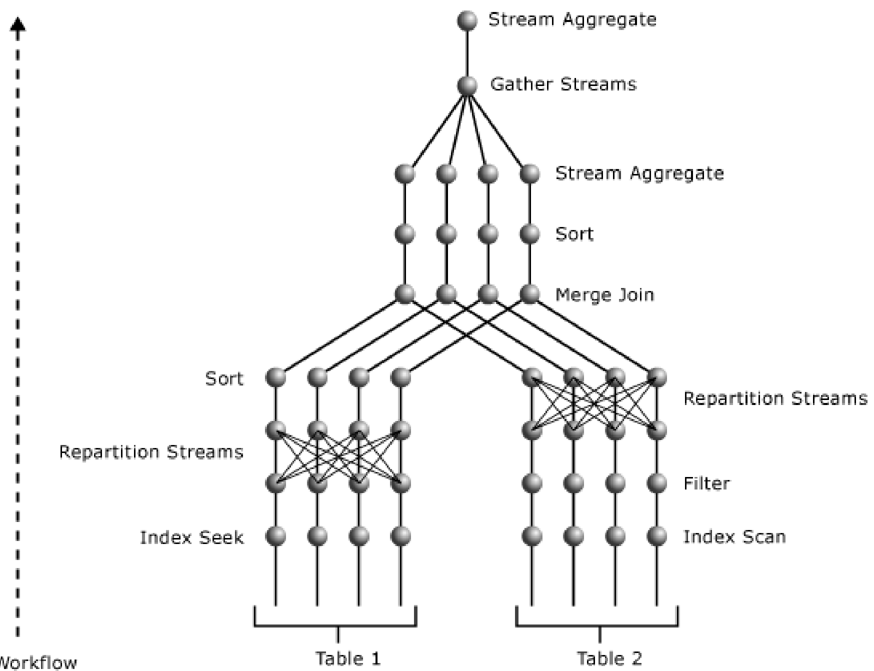
```
CREATE INDEX l_order_dates_idx
   ON lineitem
      (l_orderkey, l_receiptdate, l_commitdate, l_shipdate)

CREATE UNIQUE INDEX o_datkeyopr_idx
   ON ORDERS
      (o_orderdate, o_orderkey, o_custkey, o_orderpriority)
```

Here is one possible parallel plan generated for the query previously shown:

```
|--Stream Aggregate(GROUP BY:([ORDERS].[o_orderpriority])
                  DEFINE:([Expr1005]=COUNT(*)))
    |--Parallelism(Gather Streams, ORDER BY:
                  ([ORDERS].[o_orderpriority] ASC))
         |--Stream Aggregate(GROUP BY:
                  ([ORDERS].[o_orderpriority])
                  DEFINE:([Expr1005]=Count(*)))
              |--Sort(ORDER BY:([ORDERS].[o_orderpriority] ASC))
                   |--Merge Join(Left Semi Join, MERGE:
                  ([ORDERS].[o_orderkey])=
                        ([LINEITEM].[l_orderkey]),
                  RESIDUAL:([ORDERS].[o_orderkey]=
                        [LINEITEM].[l_orderkey]))
                        |--Sort(ORDER BY:([ORDERS].[o_orderkey] ASC))
                        |    |--Parallelism(Repartition Streams,
                           PARTITION COLUMNS:
                           ([ORDERS].[o_orderkey]))
                        |         |--Index Seek(OBJECT:
                     ([tpcd1G].[dbo].[ORDERS].[O_DATKEYOPR_IDX]),
                     SEEK:([ORDERS].[o_orderdate] >=
                           Apr  1 2000 12:00AM AND
                           [ORDERS].[o_orderdate] <
                           Jul  1 2000 12:00AM) ORDERED)
                        |--Parallelism(Repartition Streams,
                     PARTITION COLUMNS:
                     ([LINEITEM].[l_orderkey]),
                     ORDER BY:([LINEITEM].[l_orderkey] ASC))
                             |--Filter(WHERE:
                           ([LINEITEM].[l_commitdate]<
                           [LINEITEM].[l_receiptdate]))
                                  |--Index Scan(OBJECT:
         ([tpcd1G].[dbo].[LINEITEM].[L_ORDER_DATES_IDX]), ORDERED)
```

The illustration below shows a query plan executed with a degree of parallelism equal to 4 and involving a two-table join.




The parallel plan contains three parallelism operators. Both the Index Seek operator of the `o_datkey_ptr` index and the Index Scan operator of the `l_order_dates_idx` index are performed in parallel. This produces several exclusive streams. This can be determined from the nearest Parallelism operators above the Index Scan and Index Seek operators, respectively. Both are repartitioning the type of exchange. That is, they are just reshuffling data among the streams and producing the same number of streams on their output as they have on their input. This number

of streams is equal to the degree of parallelism.

The parallelism operator above the `l_order_dates_idx` Index Scan operator is repartitioning its input streams using the value of `L_ORDERKEY` as a key. In this way, the same values of `L_ORDERKEY` end up in the same output stream. At the same time, output streams maintain the order on the `L_ORDERKEY` column to meet the input requirement of the Merge Join operator.

The parallelism operator above the Index Seek operator is repartitioning its input streams using the value of `O_ORDERKEY`. Because its input is not sorted on the `O_ORDERKEY` column values and this is the join column in the `Merge Join` operator, the Sort operator between the parallelism and Merge Join operators make sure that the input is sorted for the `Merge Join` operator on the join columns. The `Sort` operator, like the Merge Join operator, is performed in parallel.

The topmost parallelism operator gathers results from several streams into a single stream. Partial aggregations performed by the Stream Aggregate operator below the parallelism operator are then accumulated into a single `SUM` value for each different value of the `O_ORDERPRIORITY` in the Stream Aggregate operator above the parallelism operator. Because this plan has two exchange segments, with degree of parallelism equal to 4, it uses eight worker threads.

For more information on the operators used in this example, refer to the Showplan Logical and Physical Operators Reference.

**Parallel Index Operations**

The query plans built for the index operations that create or rebuild an index, or drop a clustered index, allow for parallel, multi-worker threaded operations on computers that have multiple microprocessors.

> **NOTE**
>
> Parallel index operations are only available in Enterprise Edition, starting with SQL Server 2008.

SQL Server uses the same algorithms to determine the degree of parallelism (the total number of separate worker threads to run) for index operations as it does for other queries. The maximum degree of parallelism for an index operation is subject to the max degree of parallelism server configuration option. You can override the max degree of parallelism value for individual index operations by setting the MAXDOP index option in the CREATE INDEX, ALTER INDEX, DROP INDEX, and ALTER TABLE statements.

When the SQL Server Database Engine builds an index execution plan, the number of parallel operations is set to the lowest value from among the following:

- The number of microprocessors, or CPUs in the computer.
- The number specified in the max degree of parallelism server configuration option.
- The number of CPUs not already over a threshold of work performed for SQL Server worker threads.

For example, on a computer that has eight CPUs, but where max degree of parallelism is set to 6, no more than six parallel worker threads are generated for an index operation. If five of the CPUs in the computer exceed the threshold of SQL Server work when an index execution plan is built, the execution plan specifies only three parallel worker threads.

The main phases of a parallel index operation include the following:

- A coordinating worker thread quickly and randomly scans the table to estimate the distribution of the index keys. The coordinating worker thread establishes the key boundaries that will create a number of key ranges equal to the degree of parallel operations, where each key range is estimated to cover similar numbers of rows. For example, if there are four million rows in the table and the degree of parallelism is 4, the coordinating worker thread will determine the key values that delimit four sets of rows with 1 million rows in each set. If

enough key ranges cannot be established to use all CPUs, the degree of parallelism is reduced accordingly.
- The coordinating worker thread dispatches a number of worker threads equal to the degree of parallel operations and waits for these worker threads to complete their work. Each worker thread scans the base table using a filter that retrieves only rows with key values within the range assigned to the worker thread. Each worker thread builds an index structure for the rows in its key range. In the case of a partitioned index, each worker thread builds a specified number of partitions. Partitions are not shared among worker threads.
- After all the parallel worker threads have completed, the coordinating worker thread connects the index subunits into a single index. This phase applies only to offline index operations.

Individual `CREATE TABLE` or `ALTER TABLE` statements can have multiple constraints that require that an index be created. These multiple index creation operations are performed in series, although each individual index creation operation may be a parallel operation on a computer that has multiple CPUs.

# Distributed Query Architecture

Microsoft SQL Server supports two methods for referencing heterogeneous OLE DB data sources in Transact-SQL statements:

- Linked server names
  The system stored procedures `sp_addlinkedserver` and `sp_addlinkedsrvlogin` are used to give a server name to an OLE DB data source. Objects in these linked servers can be referenced in Transact-SQL statements using four-part names. For example, if a linked server name of `DeptSQLSrvr` is defined against another instance of SQL Server, the following statement references a table on that server:

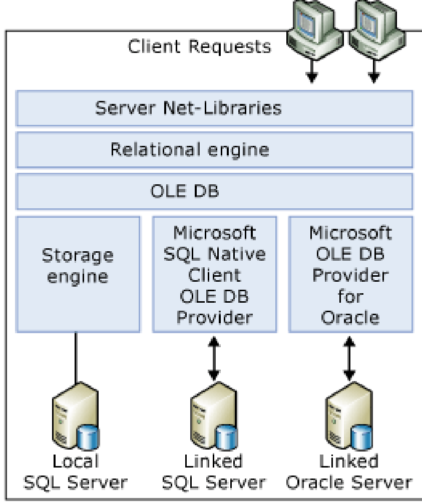
  ```
  SELECT JobTitle, HireDate
  FROM DeptSQLSrvr.AdventureWorks2014.HumanResources.Employee;
  ```

  The linked server name can also be specified in an `OPENQUERY` statement to open a rowset from the OLE DB data source. This rowset can then be referenced like a table in Transact-SQL statements.

- Ad hoc connector names
  For infrequent references to a data source, the `OPENROWSET` or `OPENDATASOURCE` functions are specified with the information needed to connect to the linked server. The rowset can then be referenced the same way a table is referenced in Transact-SQL statements:

  ```
  SELECT *
  FROM OPENROWSET('Microsoft.Jet.OLEDB.4.0',
        'c:\MSOffice\Access\Samples\Northwind.mdb';'Admin';'';
        Employees);
  ```

SQL Server uses OLE DB to communicate between the relational engine and the storage engine. The relational engine breaks down each Transact-SQL statement into a series of operations on simple OLE DB rowsets opened by the storage engine from the base tables. This means the relational engine can also open simple OLE DB rowsets on any OLE DB data source.




The relational engine uses the OLE DB application programming interface (API) to open the rowsets on linked servers, fetch the rows, and manage transactions.

For each OLE DB data source accessed as a linked server, an OLE DB provider must be present on the server running SQL Server. The set of Transact-SQL operations that can be used against a specific OLE DB data source depends on the capabilities of the OLE DB provider.

For each instance of SQL Server, members of the `sysadmin` fixed server role can enable or disable the use of ad-hoc connector names for an OLE DB provider using the SQL Server `DisallowAdhocAccess` property. When ad-hoc access is enabled, any user logged on to that instance can execute SQL statements containing ad-hoc connector names, referencing any data source on the network that can be accessed using that OLE DB provider. To control access to data sources, members of the `sysadmin` role can disable ad-hoc access for that OLE DB provider, thereby limiting users to only those data sources referenced by linked server names defined by the administrators. By default, ad-hoc access is enabled for the SQL Server OLE DB provider, and disabled for all other OLE DB providers.

Distributed queries can allow users to access another data source (for example, files, non-relational data sources such as Active Directory, and so on) using the security context of the Microsoft Windows account under which the SQL Server service is running. SQL Server impersonates the login appropriately for Windows logins; however, that is not possible for SQL Server logins. This can potentially allow a distributed query user to access another data source for which they do not have permissions, but the account under which the SQL Server service is running does have permissions. Use `sp_addlinkedsrvlogin` to define the specific logins that are authorized to access the corresponding linked server. This control is not available for ad-hoc names, so use caution in enabling an OLE DB provider for ad-hoc access.

When possible, SQL Server pushes relational operations such as joins, restrictions, projections, sorts, and group by operations to the OLE DB data source. SQL Server does not default to scanning the base table into SQL Server and performing the relational operations itself. SQL Server queries the OLE DB provider to determine the level of SQL grammar it supports, and, based on that information, pushes as many relational operations as possible to the provider.

SQL Server specifies a mechanism for an OLE DB provider to return statistics indicating how key values are distributed within the OLE DB data source. This lets the SQL Server Query Optimizer better analyze the pattern of data in the data source against the requirements of each SQL statement, increasing the ability of the Query Optimizer to generate optimal execution plans.

## Query Processing Enhancements on Partitioned Tables and Indexes

SQL Server 2008 improved query processing performance on partitioned tables for many parallel plans, changes the way parallel and serial plans are represented, and enhanced the partitioning information provided in both compile-time and run-time execution plans. This topic describes these improvements, provides guidance on how

to interpret the query execution plans of partitioned tables and indexes, and provides best practices for improving query performance on partitioned objects.

> **NOTE**
>
> Partitioned tables and indexes are supported only in the SQL Server Enterprise, Developer, and Evaluation editions.

**New Partition-Aware Seek Operation**

In SQL Server, the internal representation of a partitioned table is changed so that the table appears to the query processor to be a multicolumn index with `PartitionID` as the leading column. `PartitionID` is a hidden computed column used internally to represent the `ID` of the partition containing a specific row. For example, assume the table T, defined as `T(a, b, c)`, is partitioned on column a, and has a clustered index on column b. In SQL Server, this partitioned table is treated internally as a nonpartitioned table with the schema `T(PartitionID, a, b, c)` and a clustered index on the composite key `(PartitionID, b)`. This allows the Query Optimizer to perform seek operations based on `PartitionID` on any partitioned table or index.

Partition elimination is now done in this seek operation.

In addition, the Query Optimizer is extended so that a seek or scan operation with one condition can be done on `PartitionID` (as the logical leading column) and possibly other index key columns, and then a second-level seek, with a different condition, can be done on one or more additional columns, for each distinct value that meets the qualification for the first-level seek operation. That is, this operation, called a skip scan, allows the Query Optimizer to perform a seek or scan operation based on one condition to determine the partitions to be accessed and a second-level index seek operation within that operator to return rows from these partitions that meet a different condition. For example, consider the following query.

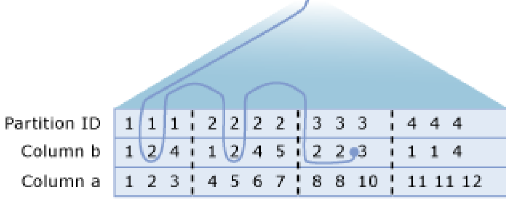
```
SELECT * FROM T WHERE a < 10 and b = 2;
```

For this example, assume that table T, defined as `T(a, b, c)`, is partitioned on column a, and has a clustered index on column b. The partition boundaries for table T are defined by the following partition function:

```
CREATE PARTITION FUNCTION myRangePF1 (int) AS RANGE LEFT FOR VALUES (3, 7, 10);
```

To solve the query, the query processor performs a first-level seek operation to find every partition that contains rows that meet the condition `T.a < 10`. This identifies the partitions to be accessed. Within each partition identified, the processor then performs a second-level seek into the clustered index on column b to find the rows that meet the condition `T.b = 2` and `T.a < 10`.

The following illustration is a logical representation of the skip scan operation. It shows table `T` with data in columns `a` and `b`. The partitions are numbered 1 through 4 with the partition boundaries shown by dashed vertical lines. A first-level seek operation to the partitions (not shown in the illustration) has determined that partitions 1, 2, and 3 meet the seek condition implied by the partitioning defined for the table and the predicate on column `a`. That is, `T.a < 10`. The path traversed by the second-level seek portion of the skip scan operation is illustrated by the curved line. Essentially, the skip scan operation seeks into each of these partitions for rows that meet the condition `b = 2`. The total cost of the skip scan operation is the same as that of three separate index seeks.



### Displaying Partitioning Information in Query Execution Plans

The execution plans of queries on partitioned tables and indexes can be examined by using the Transact-SQL `SET` statements `SET SHOWPLAN_XML` or `SET STATISTICS XML`, or by using the graphical execution plan output in SQL Server Management Studio. For example, you can display the compile-time execution plan by clicking *Display Estimated Execution Plan* on the Query Editor toolbar and the run-time plan by clicking *Include Actual Execution Plan*.

Using these tools, you can ascertain the following information:

- The operations such as `scans`, `seeks`, `inserts`, `updates`, `merges`, and `deletes` that access partitioned tables or indexes.
- The partitions accessed by the query. For example, the total count of partitions accessed and the ranges of contiguous partitions that are accessed are available in run-time execution plans.
- When the skip scan operation is used in a seek or scan operation to retrieve data from one or more partitions.

#### Partition Information Enhancements

SQL Server provides enhanced partitioning information for both compile-time and run-time execution plans. Execution plans now provide the following information:

- An optional `Partitioned` attribute that indicates that an operator, such as a `seek`, `scan`, `insert`, `update`, `merge`, or `delete`, is performed on a partitioned table.
- A new `SeekPredicateNew` element with a `SeekKeys` subelement that includes `PartitionID` as the leading index key column and filter conditions that specify range seeks on `PartitionID`. The presence of two `SeekKeys` subelements indicates that a skip scan operation on `PartitionID` is used.
- Summary information that provides a total count of the partitions accessed. This information is available only in run-time plans.

To demonstrate how this information is displayed in both the graphical execution plan output and the XML Showplan output, consider the following query on the partitioned table `fact_sales`. This query updates data in two partitions.

```
UPDATE fact_sales
SET quantity = quantity * 2
WHERE date_id BETWEEN 20080802 AND 20080902;
```

The following illustration shows the properties of the `Clustered Index Seek` operator in the compile-time execution plan for this query. To view the definition of the `fact_sales` table and the partition definition, see "Example" in this topic.

| Clustered Index Seek (Clustered) | |
|---|---|
| Scanning a particular range of rows from a clustered index. | |
| **Physical Operation** | Clustered Index Seek |
| **Logical Operation** | Clustered Index Seek |
| **Actual Execution Mode** | Row |
| **Estimated Execution Mode** | Row |
| **Storage** | RowStore |
| **Number of Rows Read** | 967333 |
| **Actual Number of Rows** | 967333 |
| **Actual Number of Batches** | 0 |
| **Estimated Operator Cost** | 102.553 (5%) |
| **Estimated I/O Cost** | 101.498 |
| **Estimated Subtree Cost** | 102.553 |
| **Estimated CPU Cost** | 1.05493 |
| **Estimated Number of Executions** | 1 |
| **Number of Executions** | 1 |
| **Estimated Number of Rows** | 958743 |
| **Estimated Number of Rows to be Read** | 958743 |
| **Estimated Row Size** | 23 B |
| **Actual Rebinds** | 0 |
| **Actual Rewinds** | 0 |
| **Partitioned** | True |
| **Actual Partition Count** | 2 |
| **Ordered** | True |
| **Node ID** | 2 |

**Object**
[db_sales_test].[dbo].[fact_sales].[ci]
**Output List**
PtnId1000, Uniq1002, [db_sales_test].[dbo].[fact_sales].date_id, [db_sales_test].[dbo].[fact_sales].quantity
**Seek Predicates**
Seek Keys[1]: Start: PtnId1000 >= Scalar Operator (RangePartitionNew([@2],(1),(20080801),(20080901),(20081001),(20081101),(20081201),(20090101))), End: PtnId1000 <= Scalar Operator(RangePartitionNew([@3],(1),(20080801),(20080901),(20081001),(20081101),(20081201),(20090101))), Seek Keys[2]: Start: [db_sales_test].[dbo].[fact_sales].date_id >= Scalar Operator ([@2]), End: [db_sales_test].[dbo].[fact_sales].date_id <= Scalar Operator([@3])

**Partitioned Attribute**

When an operator such as an `Index Seek` is executed on a partitioned table or index, the `Partitioned` attribute appears in the compile-time and run-time plan and is set to `True` (1). The attribute does not display when it is set to `False` (0).

The `Partitioned` attribute can appear in the following physical and logical operators:

- `Table Scan`
- `Index Scan`
- `Index Seek`
- `Insert`
- `Update`
- `Delete`
- `Merge`

As shown in the previous illustration, this attribute is displayed in the properties of the operator in which it is defined. In the XML Showplan output, this attribute appears as `Partitioned="1"` in the `RelOp` node of the operator in which it is defined.

**New Seek Predicate**

In XML Showplan output, the `SeekPredicateNew` element appears in the operator in which it is defined. It can contain up to two occurrences of the `SeekKeys` sub-element. The first `SeekKeys` item specifies the first-level seek operation at the partition ID level of the logical index. That is, this seek determines the partitions that must be accessed to satisfy the conditions of the query. The second `SeekKeys` item specifies the second-level seek portion

of the skip scan operation that occurs within each partition identified in the first-level seek.

#### Partition Summary Information

In run-time execution plans, partition summary information provides a count of the partitions accessed and the identity of the actual partitions accessed. You can use this information to verify that the correct partitions are accessed in the query and that all other partitions are eliminated from consideration.

The following information is provided: `Actual Partition Count`, and `Partitions Accessed`.

`Actual Partition Count` is the total number of partitions accessed by the query.

`Partitions Accessed`, in XML Showplan output, is the partition summary information that appears in the new `RuntimePartitionSummary` element in `RelOp` node of the operator in which it is defined. The following example shows the contents of the `RuntimePartitionSummary` element, indicating that two total partitions are accessed (partitions 2 and 3).

```
<RunTimePartitionSummary>

    <PartitionsAccessed PartitionCount="2" >

        <PartitionRange Start="2" End="3" />

    </PartitionsAccessed>

</RunTimePartitionSummary>
```

#### Displaying Partition Information by Using Other Showplan Methods

The Showplan methods `SHOWPLAN_ALL`, `SHOWPLAN_TEXT`, and `STATISTICS PROFILE` do not report the partition information described in this topic, with the following exception. As part of the `SEEK` predicate, the partitions to be accessed are identified by a range predicate on the computed column representing the partition ID. The following example shows the `SEEK` predicate for a `Clustered Index Seek` operator. Partitions 2 and 3 are accessed, and the seek operator filters on the rows that meet the condition `date_id BETWEEN 20080802 AND 20080902`.

```
|--Clustered Index Seek(OBJECT:([db_sales_test].[dbo].[fact_sales].[ci]),

        SEEK:([PtnId1000] >= (2) AND [PtnId1000] \<= (3)

                AND [db_sales_test].[dbo].[fact_sales].[date_id] >= (20080802)

                AND [db_sales_test].[dbo].[fact_sales].[date_id] <= (20080902))

                ORDERED FORWARD)
```

#### Interpreting Execution Plans for Partitioned Heaps

A partitioned heap is treated as a logical index on the partition ID. Partition elimination on a partitioned heap is represented in an execution plan as a `Table Scan` operator with a `SEEK` predicate on partition ID. The following example shows the Showplan information provided:

```
|-- Table Scan (OBJECT: ([db].[dbo].[T]), SEEK: ([PtnId1001]=[Expr1011]) ORDERED FORWARD)
```

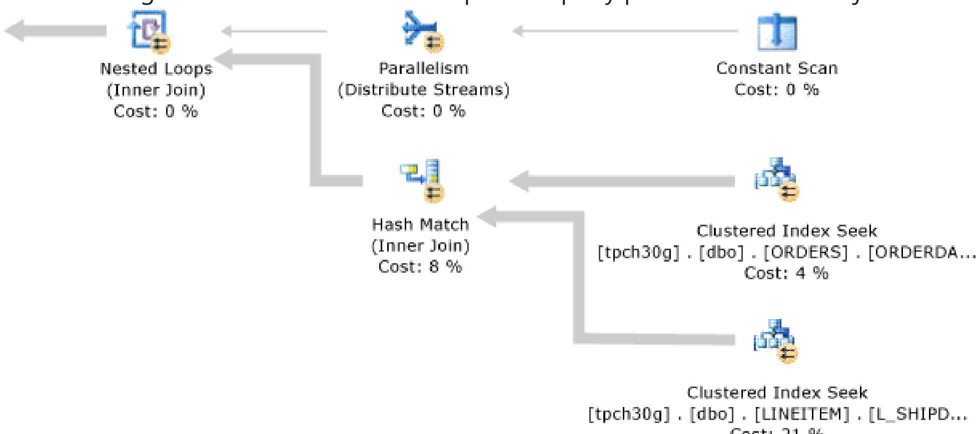
#### Interpreting Execution Plans for Collocated Joins

Join collocation can occur when two tables are partitioned using the same or equivalent partitioning function and the partitioning columns from both sides of the join are specified in the join condition of the query. The Query Optimizer can generate a plan where the partitions of each table that have equal partition IDs are joined separately. Collocated joins can be faster than non-collocated joins because they can require less memory and processing time. The Query Optimizer chooses a non-collocated plan or a collocated plan based on cost estimates.

In a collocated plan, the `Nested Loops` join reads one or more joined table or index partitions from the inner side. The numbers within the `Constant Scan` operators represent the partition numbers.

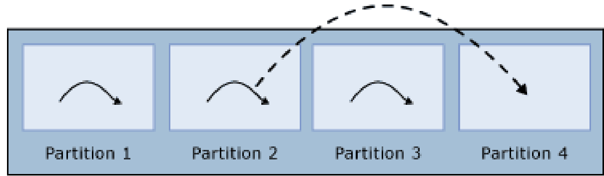
When parallel plans for collocated joins are generated for partitioned tables or indexes, a Parallelism operator appears between the `Constant Scan` and the `Nested Loops` join operators. In this case, multiple worker threads on the outer side of the join each read and work on a different partition.

The following illustration demonstrates a parallel query plan for a collocated join.

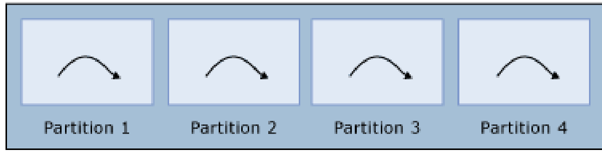



**Parallel Query Execution Strategy for Partitioned Objects**

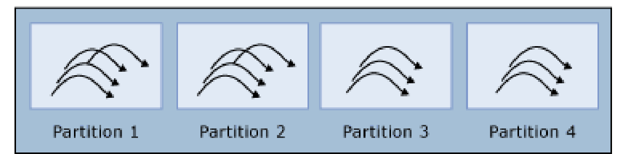
The query processor uses a parallel execution strategy for queries that select from partitioned objects. As part of the execution strategy, the query processor determines the table partitions required for the query and the proportion of worker threads to allocate to each partition. In most cases, the query processor allocates an equal or almost equal number of worker threads to each partition, and then executes the query in parallel across the partitions. The following paragraphs explain worker thread allocation in greater detail.




If the number of worker threads is less than the number of partitions, the query processor assigns each worker thread to a different partition, initially leaving one or more partitions without an assigned worker thread. When a worker thread finishes executing on a partition, the query processor assigns it to the next partition until each partition has been assigned a single worker thread. This is the only case in which the query processor reallocates worker threads to other partitions.
Shows worker thread reassigned after it finishes. If the number of worker threads is equal to the number of partitions, the query processor assigns one worker thread to each partition. When a worker thread finishes, it is not reallocated to another partition.




If the number of worker threads is greater than the number of partitions, the query processor allocates an equal number of worker threads to each partition. If the number of worker threads is not an exact multiple of the number of partitions, the query processor allocates one additional worker thread to some partitions in order to use all of the available worker threads. Note that if there is only one partition, all worker threads will be assigned to that partition. In the diagram below, there are four partitions and 14 worker threads. Each partition has 3 worker threads assigned, and two partitions have an additional worker thread, for a total of 14 worker thread assignments. When a worker thread finishes, it is not reassigned to another partition.




Although the above examples suggest a straightforward way to allocate worker threads, the actual strategy is more complex and accounts for other variables that occur during query execution. For example, if the table is partitioned and has a clustered index on column A and a query has the predicate clause `WHERE A IN (13, 17, 25)`, the query processor will allocate one or more worker threads to each of these three seek values (A=13, A=17, and A=25) instead of each table partition. It is only necessary to execute the query in the partitions that contain these values, and if all of these seek predicates happen to be in the same table partition, all of the worker threads will be assigned to the same table partition.

To take another example, suppose that the table has four partitions on column A with boundary points (10, 20, 30), an index on column B, and the query has a predicate clause `WHERE B IN (50, 100, 150)`. Because the table partitions are based on the values of A, the values of B can occur in any of the table partitions. Thus, the query processor will seek for each of the three values of B (50, 100, 150) in each of the four table partitions. The query processor will assign worker threads proportionately so that it can execute each of these 12 query scans in parallel.

| TABLE PARTITIONS BASED ON COLUMN A | SEEKS FOR COLUMN B IN EACH TABLE PARTITION |
| --- | --- |
| Table Partition 1: A < 10 | B=50, B=100, B=150 |
| Table Partition 2: A >= 10 AND A < 20 | B=50, B=100, B=150 |
| Table Partition 3: A >= 20 AND A < 30 | B=50, B=100, B=150 |
| Table Partition 4: A >= 30 | B=50, B=100, B=150 |

**Best Practices**

To improve the performance of queries that access a large amount of data from large partitioned tables and indexes, we recommend the following best practices:

- Stripe each partition across many disks. This is especially relevant when using spinning disks.
- When possible, use a server with enough main memory to fit frequently accessed partitions or all partitions in memory to reduce I/O cost.
- If the data you query will not fit in memory, compress the tables and indexes. This will reduce I/O cost.
- Use a server with fast processors and as many processor cores as you can afford, to take advantage of parallel query processing capability.
- Ensure the server has sufficient I/O controller bandwidth.
- Create a clustered index on every large partitioned table to take advantage of B-tree scanning optimizations.
- Follow the best practice recommendations in the white paper, The Data Loading Performance Guide, when bulk loading data into partitioned tables.

**Example**

The following example creates a test database containing a single table with seven partitions. Use the tools described previously when executing the queries in this example to view partitioning information for both compile-time and run-time plans.

**NOTE**

This example inserts more than 1 million rows into the table. Running this example may take several minutes depending on your hardware. Before executing this example, verify that you have more than 1.5 GB of disk space available.

```
USE master;
GO
IF DB_ID (N'db_sales_test') IS NOT NULL
    DROP DATABASE db_sales_test;
GO
CREATE DATABASE db_sales_test;
GO
USE db_sales_test;
GO
CREATE PARTITION FUNCTION [pf_range_fact](int) AS RANGE RIGHT FOR VALUES
(20080801, 20080901, 20081001, 20081101, 20081201, 20090101);
GO
CREATE PARTITION SCHEME [ps_fact_sales] AS PARTITION [pf_range_fact]
ALL TO ([PRIMARY]);
GO
CREATE TABLE fact_sales(date_id int, product_id int, store_id int,
    quantity int, unit_price numeric(7,2), other_data char(1000))
ON ps_fact_sales(date_id);
GO
CREATE CLUSTERED INDEX ci ON fact_sales(date_id);
GO
PRINT 'Loading...';
SET NOCOUNT ON;
DECLARE @i int;
SET @i = 1;
WHILE (@i<1000000)
BEGIN
    INSERT INTO fact_sales VALUES(20080800 + (@i%30) + 1, @i%10000, @i%200, RAND() * 25, (@i%3) + 1, '');
    SET @i += 1;
END;
GO
DECLARE @i int;
SET @i = 1;
WHILE (@i<10000)
BEGIN
    INSERT INTO fact_sales VALUES(20080900 + (@i%30) + 1, @i%10000, @i%200, RAND() * 25, (@i%3) + 1, '');
    SET @i += 1;
END;
PRINT 'Done.';
GO
-- Two-partition query.
SET STATISTICS XML ON;
GO
SELECT date_id, SUM(quantity*unit_price) AS total_price
FROM fact_sales
WHERE date_id BETWEEN 20080802 AND 20080902
GROUP BY date_id ;
GO
SET STATISTICS XML OFF;
GO
-- Single-partition query.
SET STATISTICS XML ON;
GO
SELECT date_id, SUM(quantity*unit_price) AS total_price
FROM fact_sales
WHERE date_id BETWEEN 20080801 AND 20080831
GROUP BY date_id;
GO
SET STATISTICS XML OFF;
GO
```

## Additional Reading

Showplan Logical and Physical Operators Reference
Extended Events

Best Practice with the Query Store
Cardinality Estimation
Adaptive query processing
Operator Precedence

# Thread and Task Architecture Guide

5/3/2018 • 8 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Threads are an operating system feature that lets application logic be separated into several concurrent execution paths. This feature is useful when complex applications have many tasks that can be performed at the same time.

When an operating system executes an instance of an application, it creates a unit called a process to manage the instance. The process has a thread of execution. This is the series of programming instructions performed by the application code. For example, if a simple application has a single set of instructions that can be performed serially, there is just one execution path or thread through the application. More complex applications may have several tasks that can be performed in tandem, instead of serially. The application can do this by starting separate processes for each task. However, starting a process is a resource-intensive operation. Instead, an application can start separate threads. These are relatively less resource-intensive. Additionally, each thread can be scheduled for execution independently from the other threads associated with a process.

Threads allow complex applications to make more effective use of a CPU, even on computers that have a single CPU. With one CPU, only one thread can execute at a time. If one thread executes a long-running operation that does not use the CPU, such as a disk read or write, another one of the threads can execute until the first operation is completed. By being able to execute threads while other threads are waiting for an operation to be completed, an application can maximize its use of the CPU. This is especially true for multi-user, disk I/O intensive applications such as a database server. Computers that have multiple microprocessors or CPUs can execute one thread per CPU at the same time. For example, if a computer has eight CPUs, it can execute eight threads at the same time.

## SQL Server Batch or Task Scheduling

**Allocating Threads to a CPU**

By default, each instance of SQL Server starts each thread. If affinity has been enabled, the operating system assigns each thread to a specific CPU. The operating system distributes threads from instances of SQL Server among the microprocessors, or CPUs on a computer based on load. Sometimes, the operating system can also move a thread from one CPU with heavy usage to another CPU. In contrast, the SQL Server Database Engine assigns worker threads to schedulers that distribute the threads evenly among the CPUs.

The affinity mask option is set by using ALTER SERVER CONFIGURATION. When the affinity mask is not set, the instance of SQL Server allocates worker threads evenly among the schedulers that have not been masked off

**Using the lightweight pooling Option**

The overhead involved in switching thread contexts is not very large. Most instances of SQL Server will not see any performance differences between setting the lightweight pooling option to 0 or 1. The only instances of SQL Server that might benefit from lightweight pooling are those that run on a computer having the following characteristics:

- A large multi-CPU server.
- All the CPUs are running near maximum capacity.
- There is a high level of context switching.

These systems may see a small increase in performance if the lightweight pooling value is set to 1.

We do not recommend that you use fiber mode scheduling for routine operation. This is because it can decrease performance by inhibiting the regular benefits of context switching, and because some components of SQL Server

cannot function correctly in fiber mode. For more information, see lightweight pooling.

## Thread and Fiber Execution

Microsoft Windows uses a numeric priority system that ranges from 1 through 31 to schedule threads for execution. Zero is reserved for operating system use. When several threads are waiting to execute, Windows dispatches the thread with the highest priority.

By default, each instance of SQL Server is a priority of 7, which is referred to as the normal priority. This default gives SQL Server threads a high enough priority to obtain sufficient CPU resources without adversely affecting other applications.

The priority boost configuration option can be used to increase the priority of the threads from an instance of SQL Server to 13. This is referred to as high priority. This setting gives SQL Server threads a higher priority than most other applications. Thus, SQL Server threads will generally be dispatched whenever they are ready to run and will not be pre-empted by threads from other applications. This can improve performance when a server is running only instances of SQL Server and no other applications. However, if a memory-intensive operation occurs in SQL Server, however, other applications are not likely to have a high-enough priority to pre-empt the SQL Server thread.

If you are running multiple instances of SQL Server on a computer, and turn on priority boost for only some of the instances, the performance of any instances running at normal priority can be adversely affected. Also, the performance of other applications and components on the server can decline if priority boost is turned on. Therefore, it should only be used under tightly controlled conditions.

## Hot Add CPU

Hot add CPU is the ability to dynamically add CPUs to a running system. Adding CPUs can occur physically by adding new hardware, logically by online hardware partitioning, or virtually through a virtualization layer. Starting with SQL Server 2008, SQL Server supports hot add CPU.

Requirements for hot add CPU:

- Requires hardware that supports hot add CPU.
- Requires the 64-bit edition of Windows Server 2008 Datacenter or the Windows Server 2008 Enterprise Edition for Itanium-Based Systems operating system.
- Requires SQL Server Enterprise.
- SQL Server cannot be configured to use soft NUMA. For more information about soft NUMA, see Soft-NUMA (SQL Server).

SQL Server does not automatically start to use CPUs after they are added. This prevents SQL Server from using CPUs that might be added for some other purpose. After adding CPUs, execute the RECONFIGURE statement, so that SQL Server will recognize the new CPUs as available resources.

> **NOTE**
>
> If the affinity64 mask is configured, the affinity64 mask must be modified to use the new CPUs.

## Best Practices for Running SQL Server on Computers That Have More Than 64 CPUs

**Assigning Hardware Threads with CPUs**

Do not use the affinity mask and affinity64 mask server configuration options to bind processors to specific threads. These options are limited to 64 CPUs. Use SET PROCESS AFFINITY option of ALTER SERVER

CONFIGURATION instead.

### Managing the Transaction Log File Size

Do not rely on autogrow to increase the size of the transaction log file. Increasing the transaction log must be a serial process. Extending the log can prevent transaction write operations from proceeding until the log extension is finished. Instead, preallocate space for the log files by setting the file size to a value large enough to support the typical workload in the environment.

### Setting Max Degree of Parallelism for Index Operations

The performance of index operations such as creating or rebuilding indexes can be improved on computers that have many CPUs by temporarily setting the recovery model of the database to either the bulk-logged or simple recovery model. These index operations can generate significant log activity and log contention can affect the best degree of parallelism (DOP) choice made by SQL Server.

In addition, consider adjusting the **max degree of parallelism (MAXDOP)** server configuration option for these operations. The following guidelines are based on internal tests and are general recommendations. You should try several different MAXDOP settings to determine the optimal setting for your environment.

- For the full recovery model, limit the value of the max degree of parallelism option to eight or less.
- For the bulk-logged model or the simple recovery model, setting the value of the max degree of parallelism option to a value higher than eight should be considered.
- For servers that have NUMA configured, the maximum degree of parallelism should not exceed the number of CPUs that are assigned to each NUMA node. This is because the query is more likely to use local memory from 1 NUMA node, which can improve memory access time.
- For servers that have hyper-threading enabled and were manufactured in 2009 or earlier (before hyper-threading feature was improved), the MAXDOP value should not exceed the number of physical processors, rather than logical processors.

For more information about the max degree of parallelism option, see Configure the max degree of parallelism Server Configuration Option.

### Setting the Maximum Number of Worker Threads

Always set the maximum number of worker threads to be more than the setting for the maximum degree of parallelism. The number of worker threads must always be set to a value of at least seven times the number of CPUs that are present on the server. For more information, see Configure the max worker threads Option.

### Using SQL Trace and SQL Server Profiler

We recommend that you do not use SQL Trace and SQL Server Profiler in a production environment. The overhead for running these tools also increases as the number of CPUs increases. If you must use SQL Trace in a production environment, limit the number of trace events to a minimum. Carefully profile and test each trace event under load, and avoid using combinations of events that significantly affect performance.

### Setting the Number of tempdb Data Files

Typically, the number of tempdb data files should match the number of CPUs. However, by carefully considering the concurrency needs of tempdb, you can reduce database management. For example, if a system has 64 CPUs and usually only 32 queries use tempdb, increasing the number of tempdb files to 64 will not improve performance.

### SQL Server Components That Can Use More Than 64 CPUs

The following table lists SQL Server components and indicates whether they can use more that 64 CPUs.

| PROCESS NAME | EXECUTABLE PROGRAM | USE MORE THAN 64 CPUS |
|---|---|---|
| SQL Server Database Engine | Sqlserver.exe | Yes |

| PROCESS NAME | EXECUTABLE PROGRAM | USE MORE THAN 64 CPUS |
| --- | --- | --- |
| Reporting Services | Rs.exe | No |
| Analysis Services | As.exe | No |
| Integration Services | Is.exe | No |
| Service Broker | Sb.exe | No |
| Full-Text Search | Fts.exe | No |
| SQL Server Agent | Sqlagent.exe | No |
| SQL Server Management Studio | Ssms.exe | No |
| SQL Server Setup | Setup.exe | No |

# SQL Server Transaction Log Architecture and Management Guide

5/3/2018 • 20 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Every SQL Server database has a transaction log that records all transactions and the database modifications that are made by each transaction. The transaction log is a critical component of the database and, if there is a system failure, the transaction log might be required to bring your database back to a consistent state. This guide provides information about the physical and logical architecture of the transaction log. Understanding the architecture can improve your effectiveness in managing transaction logs.

## Transaction Log Logical Architecture

The SQL Server transaction log operates logically as if the transaction log is a string of log records. Each log record is identified by a log sequence number (LSN). Each new log record is written to the logical end of the log with an LSN that is higher than the LSN of the record before it. Log records are stored in a serial sequence as they are created. Each log record contains the ID of the transaction that it belongs to. For each transaction, all log records associated with the transaction are individually linked in a chain using backward pointers that speed the rollback of the transaction.

Log records for data modifications record either the logical operation performed or they record the before and after images of the modified data. The before image is a copy of the data before the operation is performed; the after image is a copy of the data after the operation has been performed.

The steps to recover an operation depend on the type of log record:

- Logical operation logged
  - To roll the logical operation forward, the operation is performed again.
  - To roll the logical operation back, the reverse logical operation is performed.
- Before and after image logged
  - To roll the operation forward, the after image is applied.
  - To roll the operation back, the before image is applied.

Many types of operations are recorded in the transaction log. These operations include:

- The start and end of each transaction.
- Every data modification (insert, update, or delete). This includes changes by system stored procedures or data definition language (DDL) statements to any table, including system tables.
- Every extent and page allocation or deallocation.
- Creating or dropping a table or index.

  Rollback operations are also logged. Each transaction reserves space on the transaction log to make sure that enough log space exists to support a rollback that is caused by either an explicit rollback statement or if an error is encountered. The amount of space reserved depends on the operations performed in the transaction, but generally it is equal to the amount of space used to log each operation. This reserved space

is freed when the transaction is completed.

The section of the log file from the first log record that must be present for a successful database-wide rollback to the last-written log record is called the active part of the log, or the *active log*. This is the section of the log required to a full recovery of the database. No part of the active log can ever be truncated. The log sequence number (LSN) of this first log record is known as the **minimum recovery LSN (*MinLSN*)**.

# Transaction Log Physical Architecture

The transaction log in a database maps over one or more physical files. Conceptually, the log file is a string of log records. Physically, the sequence of log records is stored efficiently in the set of physical files that implement the transaction log. There must be at least one log file for each database.

The SQL Server Database Engine divides each physical log file internally into a number of virtual log files (VLFs). Virtual log files have no fixed size, and there is no fixed number of virtual log files for a physical log file. The Database Engine chooses the size of the virtual log files dynamically while it is creating or extending log files. The Database Engine tries to maintain a small number of virtual files. The size of the virtual files after a log file has been extended is the sum of the size of the existing log and the size of the new file increment. The size or number of virtual log files cannot be configured or set by administrators.

**NOTE**

Virtual log file (VLF) creation follows this method:

- If the next growth is less than 1/8 of current log physical size, then create 1 VLF that covers the growth size (Starting with SQL Server 2014 (12.x))
- If the next growth is more than 1/8 of the current log size, then use the pre-2014 method:
  - If growth is less than 64MB, create 4 VLFs that cover the growth size (e.g. for 1 MB growth, create four 256KB VLFs)
  - If growth is from 64MB up to 1GB, create 8 VLFs that cover the growth size (e.g. for 512MB growth, create eight 64MB VLFs)
  - If growth is larger than 1GB, create 16 VLFs that cover the growth size (e.g. for 8 GB growth, create sixteen 512MB VLFs)

If the log files grow to a large size in many small increments, they will have many virtual log files. **This can slow down database startup and also log backup and restore operations.** Conversely, if the log files are set to a large size with few or just one increment, they will have few very large virtual log files. For more information on properly estimating the **required size** and **autogrow** setting of a transaction log, refer to the *Recommendations* section of Manage the size of the transaction log file.

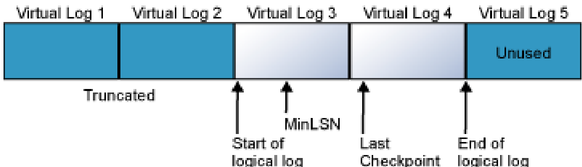
We recommend that you assign log files a *size* value close to the final size required, using the required increments to achieve optimal VLF distribution, and also have a relatively large *growth_increment* value. See the tip below to determine the optimal VLF distribution for the current transaction log size.

- The *size* value, as set by the `SIZE` argument of `ALTER DATABASE` is the initial size for the log file.
- The *growth_increment* value (also referred as the autogrow value), as set by the `FILEGROWTH` argument of `ALTER DATABASE`, is the amount of space added to the file every time new space is required.

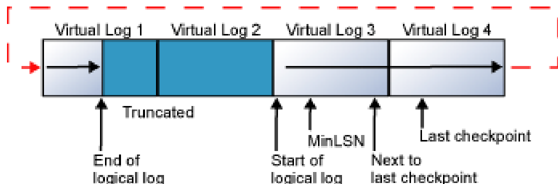
For more information on `FILEGROWTH` and `SIZE` arguments of `ALTER DATABASE`, see ALTER DATABASE (Transact-SQL) File and Filegroup Options.

> **TIP**
>
> To determine the optimal VLF distribution for the current transaction log size of all databases in a given instance, and the required growth increments to achieve the required size, see this script.

The transaction log is a wrap-around file. For example, consider a database with one physical log file divided into four VLFs. When the database is created, the logical log file begins at the start of the physical log file. New log records are added at the end of the logical log and expand toward the end of the physical log. Log truncation frees any virtual logs whose records all appear in front of the minimum recovery log sequence number (MinLSN). The *MinLSN* is the log sequence number of the oldest log record that is required for a successful database-wide rollback. The transaction log in the example database would look similar to the one in the following illustration.




When the end of the logical log reaches the end of the physical log file, the new log records wrap around to the start of the physical log file.




This cycle repeats endlessly, as long as the end of the logical log never reaches the beginning of the logical log. If the old log records are truncated frequently enough to always leave sufficient room for all the new log records created through the next checkpoint, the log never fills. However, if the end of the logical log does reach the start of the logical log, one of two things occurs:

- If the `FILEGROWTH` setting is enabled for the log and space is available on the disk, the file is extended by the amount specified in the *growth_increment* parameter and the new log records are added to the extension. For more information about the `FILEGROWTH` setting, see ALTER DATABASE File and Filegroup Options (Transact-SQL).

- If the `FILEGROWTH` setting is not enabled, or the disk that is holding the log file has less free space than the amount specified in *growth_increment*, an 9002 error is generated. Refer to Troubleshoot a Full Transaction Log for more information.

  If the log contains multiple physical log files, the logical log will move through all the physical log files before it wraps back to the start of the first physical log file.

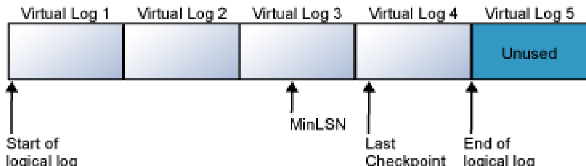
> **IMPORTANT**
>
> For more information about transaction log size management, see Manage the Size of the Transaction Log File.

### Log Truncation

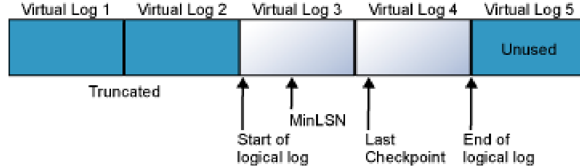
Log truncation is essential to keep the log from filling. Log truncation deletes inactive virtual log files from the logical transaction log of a SQL Server database, freeing space in the logical log for reuse by the physical transaction log. If a transaction log were never truncated, it would eventually fill all the disk space that is allocated to its physical log files. However, before the log can be truncated, a checkpoint operation must occur. A checkpoint writes the current in-memory modified pages (known as dirty pages) and transaction log information from memory to disk. When the checkpoint is performed, the inactive portion of the transaction log is marked as

reusable. Thereafter, the inactive portion can be freed by log truncation. For more information about checkpoints, see Database Checkpoints (SQL Server).

The following illustrations show a transaction log before and after truncation. The first illustration shows a transaction log that has never been truncated. Currently, four virtual log files are in use by the logical log. The logical log starts at the front of the first virtual log file and ends at virtual log 4. The MinLSN record is in virtual log 3. Virtual log 1 and virtual log 2 contain only inactive log records. These records can be truncated. Virtual log 5 is still unused and is not part of the current logical log.




The second illustration shows how the log appears after being truncated. Virtual log 1 and virtual log 2 have been freed for reuse. The logical log now starts at the beginning of virtual log 3. Virtual log 5 is still unused, and it is not part of the current logical log.




Log truncation occurs automatically after the following events, except when delayed for some reason:

- Under the simple recovery model, after a checkpoint.
- Under the full recovery model or bulk-logged recovery model, after a log backup, if a checkpoint has occurred since the previous backup.

  Log truncation can be delayed by a variety of factors. In the event of a long delay in log truncation, the transaction log can fill up. For information, see Factors that can delay log truncation and Troubleshoot a Full Transaction Log (SQL Server Error 9002).

## Write-Ahead Transaction Log

This section describes the role of the write-ahead transaction log in recording data modifications to disk. SQL Server uses a write-ahead logging (WAL) algorithm, which guarantees that no data modifications are written to disk before the associated log record is written to disk. This maintains the ACID properties for a transaction.

To understand how the write-ahead log works, it is important for you to know how modified data is written to disk. SQL Server maintains a buffer cache into which it reads data pages when data must be retrieved. When a page is modified in the buffer cache, it is not immediately written back to disk; instead, the page is marked as *dirty*. A data page can have more than one logical write made before it is physically written to disk. For each logical write, a transaction log record is inserted in the log cache that records the modification. The log records must be written to disk before the associated dirty page is removed from the buffer cache and written to disk. The checkpoint process periodically scans the buffer cache for buffers with pages from a specified database and writes all dirty pages to disk. Checkpoints save time during a later recovery by creating a point at which all dirty pages are guaranteed to have been written to disk.

Writing a modified data page from the buffer cache to disk is called flushing the page. SQL Server has logic that prevents a dirty page from being flushed before the associated log record is written. Log records are written to disk when the transactions are committed.

## Transaction Log Backups

This section presents concepts about how to back up and restore (apply) transaction logs. Under the full and bulk-logged recovery models, taking routine backups of transaction logs (*log backups*) is necessary for recovering data. You can back up the log while any full backup is running. For more information about recovery models, see Back Up and Restore of SQL Server Databases.

Before you can create the first log backup, you must create a full backup, such as a database backup or the first in a set of file backups. Restoring a database by using only file backups can become complex. Therefore, we recommend that you start with a full database backup when you can. Thereafter, backing up the transaction log regularly is necessary. This not only minimizes work-loss exposure but also enables truncation of the transaction log. Typically, the transaction log is truncated after every conventional log backup.

> **IMPORTANT**
>
> We recommend taking frequent enough log backups to support your business requirements, specifically your tolerance for work loss such as might be caused by a damaged log storage. The appropriate frequency for taking log backups depends on your tolerance for work-loss exposure balanced by how many log backups you can store, manage, and, potentially, restore. Think about the required RTO and RPO when implementing your recovery strategy, and specifically the log backup cadence. Taking a log backup every 15 to 30 minutes might be enough. If your business requires that you minimize work-loss exposure, consider taking log backups more frequently. More frequent log backups have the added advantage of increasing the frequency of log truncation, resulting in smaller log files.

> **IMPORTANT**
>
> To limit the number of log backups that you need to restore, it is essential to routinely back up your data. For example, you might schedule a weekly full database backup and daily differential database backups.
> Again, think about the required RTO and RPO when implementing your recovery strategy, and specifically the full and differential database backup cadence.

For more information about transaction log backups, see Transaction Log Backups (SQL Server).

#### The Log Chain

A continuous sequence of log backups is called a *log chain*. A log chain starts with a full backup of the database. Usually, a new log chain is only started when the database is backed up for the first time or after the recovery model is switched from simple recovery to full or bulk-logged recovery. Unless you choose to overwrite existing backup sets when creating a full database backup, the existing log chain remains intact. With the log chain intact, you can restore your database from any full database backup in the media set, followed by all subsequent log backups up through your recovery point. The recovery point could be the end of the last log backup or a specific recovery point in any of the log backups. For more information, see Transaction Log Backups (SQL Server).

To restore a database up to the point of failure, the log chain must be intact. That is, an unbroken sequence of transaction log backups must extend up to the point of failure. Where this sequence of log must start depends on the type of data backups you are restoring: database, partial, or file. For a database or partial backup, the sequence of log backups must extend from the end of a database or partial backup. For a set of file backups, the sequence of log backups must extend from the start of a full set of file backups. For more information, see Apply Transaction Log Backups (SQL Server).

#### Restore Log Backups

Restoring a log backup rolls forward the changes that were recorded in the transaction log to re-create the exact state of the database at the time the log backup operation started. When you restore a database, you will have to restore the log backups that were created after the full database backup that you restore, or from the start of the first file backup that you restore. Typically, after you restore the most recent data or differential backup, you must restore a series of log backups until you reach your recovery point. Then, you recover the database. This rolls back all transactions that were incomplete when the recovery started and brings the database online. After the database

has been recovered, you cannot restore any more backups. For more information, see Apply Transaction Log Backups (SQL Server).

## Checkpoints and the Active Portion of the Log

Checkpoints flush dirty data pages from the buffer cache of the current database to disk. This minimizes the active portion of the log that must be processed during a full recovery of a database. During a full recovery, the following types of actions are performed:

- The log records of modifications not flushed to disk before the system stopped are rolled forward.
- All modifications associated with incomplete transactions, such as transactions for which there is no COMMIT or ROLLBACK log record, are rolled back.

### Checkpoint Operation

A checkpoint performs the following processes in the database:

- Writes a record to the log file, marking the start of the checkpoint.
- Stores information recorded for the checkpoint in a chain of checkpoint log records.

  One piece of information recorded in the checkpoint is the log sequence number (LSN) of the first log record that must be present for a successful database-wide rollback. This LSN is called the Minimum Recovery LSN (MinLSN). The MinLSN is the minimum of the:

  - LSN of the start of the checkpoint.
  - LSN of the start of the oldest active transaction.
  - LSN of the start of the oldest replication transaction that has not yet been delivered to the distribution database.

    The checkpoint records also contain a list of all the active transactions that have modified the database.

- If the database uses the simple recovery model, marks for reuse the space that precedes the MinLSN.
- Writes all dirty log and data pages to disk.
- Writes a record marking the end of the checkpoint to the log file.
- Writes the LSN of the start of this chain to the database boot page.

#### Activities that cause a Checkpoint

Checkpoints occur in the following situations:

- A CHECKPOINT statement is explicitly executed. A checkpoint occurs in the current database for the connection.
- A minimally logged operation is performed in the database; for example, a bulk-copy operation is performed on a database that is using the Bulk-Logged recovery model.
- Database files have been added or removed by using ALTER DATABASE.
- An instance of SQL Server is stopped by a SHUTDOWN statement or by stopping the SQL Server (MSSQLSERVER) service. Either action causes a checkpoint in each database in the instance of SQL Server.
- An instance of SQL Server periodically generates automatic checkpoints in each database to reduce the time that the instance would take to recover the database.
- A database backup is taken.
- An activity requiring a database shutdown is performed. For example, AUTO_CLOSE is ON and the last user connection to the database is closed, or a database option change is made that requires a restart of the database.

### Automatic Checkpoints

The SQL Server Database Engine generates automatic checkpoints. The interval between automatic checkpoints is based on the amount of log space used and the time elapsed since the last checkpoint. The time interval between automatic checkpoints can be highly variable and long, if few modifications are made in the database. Automatic checkpoints can also occur frequently if lots of data is modified.

Use the **recovery interval** server configuration option to calculate the interval between automatic checkpoints for all the databases on a server instance. This option specifies the maximum time the Database Engine should use to recover a database during a system restart. The Database Engine estimates how many log records it can process in the **recovery interval** during a recovery operation.

The interval between automatic checkpoints also depends on the recovery model:

- If the database is using either the full or bulk-logged recovery model, an automatic checkpoint is generated whenever the number of log records reaches the number the Database Engine estimates it can process during the time specified in the recovery interval option.
- If the database is using the simple recovery model, an automatic checkpoint is generated whenever the number of log records reaches the lesser of these two values:
  - The log becomes 70 percent full.
  - The number of log records reaches the number the Database Engine estimates it can process during the time specified in the recovery interval option.

For information about setting the recovery interval, see Configure the recovery interval Server Configuration Option.

> **TIP**
>
> The -k SQL Server advanced setup option enables a database administrator to throttle checkpoint I/O behavior based on the throughput of the I/O subsystem for some types of checkpoints. The -k setup option applies to automatic checkpoints and any otherwise unthrottled checkpoints.

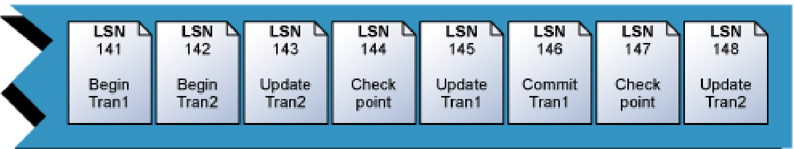
Automatic checkpoints truncate the unused section of the transaction log if the database is using the simple recovery model. However, if the database is using the full or bulk-logged recovery models, the log is not truncated by automatic checkpoints. For more information, see The Transaction Log.

The CHECKPOINT statement now provides an optional checkpoint_duration argument that specifies the requested period of time, in seconds, for checkpoints to finish. For more information, see CHECKPOINT.

### Active Log

The section of the log file from the MinLSN to the last-written log record is called the active portion of the log, or the active log. This is the section of the log required to do a full recovery of the database. No part of the active log can ever be truncated. All log records must be truncated from the parts of the log before the MinLSN.

The following illustration shows a simplified version of the end-of-a-transaction log with two active transactions. Checkpoint records have been compacted to a single record.




LSN 148 is the last record in the transaction log. At the time that the recorded checkpoint at LSN 147 was processed, Tran 1 had been committed and Tran 2 was the only active transaction. That makes the first log record for Tran 2 the oldest log record for a transaction active at the time of the last checkpoint. This makes LSN 142, the Begin transaction record for Tran 2, the MinLSN.

#### Long-Running Transactions

The active log must include every part of all uncommitted transactions. An application that starts a transaction and does not commit it or roll it back prevents the Database Engine from advancing the MinLSN. This can cause two types of problems:

- If the system is shut down after the transaction has performed many uncommitted modifications, the recovery phase of the subsequent restart can take much longer than the time specified in the **recovery interval** option.
- The log might grow very large, because the log cannot be truncated past the MinLSN. This occurs even if the database is using the simple recovery model, in which the transaction log is generally truncated on each automatic checkpoint.

#### Replication Transactions

The Log Reader Agent monitors the transaction log of each database configured for transactional replication, and it copies the transactions marked for replication from the transaction log into the distribution database. The active log must contain all transactions that are marked for replication, but that have not yet been delivered to the distribution database. If these transactions are not replicated in a timely manner, they can prevent the truncation of the log. For more information, see Transactional Replication.

## See also

We recommend the following articles and books for additional information about the transaction log and log management best practices.

The Transaction Log (SQL Server)
Manage the size of the transaction log file
Transaction Log Backups (SQL Server)
Database Checkpoints (SQL Server)
Configure the recovery interval Server Configuration Option
sys.dm_db_log_info (Transact-SQL)
sys.dm_db_log_space_usage (Transact-SQL)
Understanding Logging and Recovery in SQL Server by Paul Randal
SQL Server Transaction Log Management by Tony Davis and Gail Shaw

# Transaction Locking and Row Versioning Guide

5/3/2018 • 123 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

In any database, mismanagement of transactions often leads to contention and performance problems in systems that have many users. As the number of users that access the data increases, it becomes important to have applications that use transactions efficiently. This guide describes the locking and row versioning mechanisms the SQL Server Database Engine uses to ensure the physical integrity of each transaction and provides information on how applications can control transactions efficiently.

**Applies to**: SQL Server 2005 through SQL Server 2017, unless noted otherwise.

## 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. This is referred to as serializability because it results in the ability to reload the starting data and replay a series of transactions to end up with the data in the same state it was in after the original transactions were performed.

**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. SQL Server 2014 (12.x) and later enable delayed durable transactions. Delayed durable transactions commit before the transaction log record is persisted to disk. For more information on delayed transaction durability see the topic Transaction Durability.

SQL programmers are responsible for starting and ending transactions at points that enforce the logical consistency of the data. The programmer must define the sequence of data modifications that leave the data in a consistent state relative to the organization's business rules. The programmer includes these modification statements in a single transaction so that the SQL Server Database Engine can enforce the physical integrity of the transaction.

It is the responsibility of an enterprise database system, such as an instance of the SQL Server Database Engine, to provide mechanisms ensuring the physical integrity of each transaction. The SQL Server Database Engine provides:

- Locking facilities that preserve transaction isolation.
- Logging facilities ensure transaction durability. For fully durable transactions the log record is hardened to disk before the transactions commits. Thus, even if the server hardware, operating system, or the instance of the SQL Server Database Engine itself fails, the instance uses the transaction logs upon restart to automatically roll back any uncompleted transactions to the point of the system failure. Delayed durable transactions commit before the transaction log record is hardened to disk. Such transactions may be lost if there is a system failure before the log record is hardened to disk. For more information on delayed transaction durability see the topic Transaction Durability.
- Transaction management features that enforce transaction atomicity and consistency. After a transaction has started, it must be successfully completed (committed), or the SQL Server Database Engine undoes all of the data modifications made since the transaction started. This operation is referred to as rolling back a transaction because it returns the data to the state it was prior to those changes.

### Controlling Transactions

Applications control transactions mainly by specifying when a transaction starts and ends. This can be specified by using either Transact-SQL statements or database application programming interface (API) functions. The system must also be able to correctly handle errors that terminate a transaction before it completes. For more information, see Transactions, Transactions in ODBC and Transactions in SQL Server Native Client (OLEDB).

By default, transactions are managed at the connection level. When a transaction is started on a connection, all Transact-SQL statements executed on that connection are part of the transaction until the transaction ends. However, under a multiple active result set (MARS) session, a Transact-SQL explicit or implicit transaction becomes a batch-scoped transaction that is managed at the batch level. When the batch completes, if the batch-scoped transaction is not committed or rolled back, it is automatically rolled back by SQL Server. For more information, see Using Multiple Active Result Sets (MARS).

#### Starting Transactions

Using API functions and Transact-SQL statements, you can start transactions in an instance of the SQL Server Database Engine as explicit, autocommit, or implicit 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.

You can use all Transact-SQL statements in an explicit transaction, except for the following statements:

| | | |
|---|---|---|
| ALTER DATABASE | CREATE DATABASE | DROP FULLTEXT INDEX |
| ALTER FULLTEXT CATALOG | CREATE FULLTEXT CATALOG | RECONFIGURE |
| ALTER FULLTEXT INDEX | CREATE FULLTEXT INDEX | RESTORE |
| BACKUP | DROP DATABASE | Full-text system stored procedures |
| CREATE DATABASE | DROP FULLTEXT CATALOG | sp_dboption to set database options or any system procedure that modifies the master database inside explicit or implicit transactions. |

> **NOTE**
> UPDATE STATISTICS can be used inside an explicit transaction. However, UPDATE STATISTICS commits independently of the enclosing transaction and cannot be rolled back.

**Autocommit 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. A connection to an instance of the SQL Server Database Engine operates in autocommit mode whenever this default mode has not been overridden by either explicit or implicit transactions. Autocommit mode is also the default mode for ADO, OLE DB, ODBC, and DB-Library.

**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.

After implicit transaction mode has been set on for a connection, the instance of the SQL Server Database Engine automatically starts a transaction when it first executes any of these statements:

| | | |
|---|---|---|
| ALTER TABLE | FETCH | REVOKE |
| CREATE | GRANT | SELECT |
| DELETE | INSERT | TRUNCATE TABLE |
| DROP | OPEN | UPDATE |

**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. For more information, see the MS DTC documentation.

A transaction within a single instance of the SQL Server Database Engine that spans two or more databases is actually a distributed transaction. The instance manages the distributed transaction internally; to the user, it operates as a local 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. SQL Server 2014 (12.x) introduced delayed transaction durability. Delayed durable transactions commit before log images for the transaction are flushed to disk. For more information on delayed transaction durability see the topic Transaction Durability.

#### 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.

SQL Server Database Engine applications can manage distributed transactions either through Transact-SQL or the database API. For more information, see BEGIN DISTRIBUTED TRANSACTION (Transact-SQL).

##### 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.

> **NOTE**
>
> Under connections enabled to support multiple active result sets (MARS), an explicit transaction started through an API function cannot be committed while there are pending requests for execution. Any attempt to commit this type of transaction while there are outstanding operations running will result in an error.

##### 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. If the client's network connection to an instance of the SQL Server Database Engine is broken, any outstanding transactions for the connection are rolled back when the network notifies the instance of the break. If the client application fails or if the client computer goes down or is restarted, this also breaks the connection, and the instance of the SQL Server Database Engine rolls back any outstanding connections when the network notifies it of the break. If the client logs off the application, any outstanding transactions are rolled back.

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`. For more information, see SET XACT_ABORT (Transact-SQL).

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. For more

information with examples that include transactions, see TRY...CATCH (Transact-SQL). Beginning with SQL Server 2012 (11.x), you can use the `THROW` statement to raise an exception and transfers execution to a `CATCH` block of a `TRY...CATCH` construct. For more information, see THROW (Transact-SQL).

###### 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. In the following example, none of the `INSERT` statements in the third batch are executed because of a compile error. It appears that the first two `INSERT` statements are rolled back when they are never executed.

```
CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3));
GO
INSERT INTO TestBatch VALUES (1, 'aaa');
INSERT INTO TestBatch VALUES (2, 'bbb');
INSERT INTO TestBatch VALUSE (3, 'ccc');  -- Syntax error.
GO
SELECT * FROM TestBatch;  -- Returns no rows.
GO
```

In the following example, the third `INSERT` statement generates a run-time duplicate primary key error. The first two `INSERT` statements are successful and committed, so they remain after the run-time error.

```
CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3));
GO
INSERT INTO TestBatch VALUES (1, 'aaa');
INSERT INTO TestBatch VALUES (2, 'bbb');
INSERT INTO TestBatch VALUES (1, 'ccc');  -- Duplicate key error.
GO
SELECT * FROM TestBatch;  -- Returns rows 1 and 2.
GO
```

The SQL Server Database Engine uses deferred name resolution, in which object names are not resolved until execution time. In the following example, the first two `INSERT` statements are executed and committed, and those two rows remain in the `TestBatch` table after the third `INSERT` statement generates a run-time error by referring to a table that does not exist.

```
CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3));
GO
INSERT INTO TestBatch VALUES (1, 'aaa');
INSERT INTO TestBatch VALUES (2, 'bbb');
INSERT INTO TestBch VALUES (3, 'ccc');  -- Table name error.
GO
SELECT * FROM TestBatch;  -- Returns rows 1 and 2.
GO
```

## 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.

  Applications can choose transaction isolation levels, which define the level of protection for the transaction from modifications made by other transactions. Table-level hints can be specified for individual Transact-SQL statements to further tailor behavior to fit the requirements of the application.

### 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

Users modifying data can affect other users who are reading or modifying the same data at the same time. These users are said to be accessing the data concurrently. 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.

  For example, two editors make an electronic copy of the same document. Each editor changes the copy independently and then saves the changed copy thereby overwriting the original document. The editor who saves the changed copy last overwrites the changes made by the other editor. This problem could be avoided if one editor could not access the file until the other editor had finished and committed the transaction.

- **Uncommitted dependency (dirty read)**

  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.

  For example, an editor is making changes to an electronic document. During the changes, a second editor takes a copy of the document that includes all the changes made so far, and distributes the document to the intended audience. The first editor then decides the changes made so far are wrong and removes the edits and saves the document. The distributed document contains edits that no longer exist and should be treated as if they never existed. This problem could be avoided if no one could read the changed document until the first editor does the final save of modifications and commits the transaction.

- **Inconsistent analysis (nonrepeatable read)**

  Inconsistent analysis occurs when a second transaction accesses the same row several times and reads different data each time. Inconsistent analysis is similar to uncommitted dependency in that another

transaction is changing the data that a second transaction is reading. However, in inconsistent analysis, the data read by the second transaction was committed by the transaction that made the change. Also, inconsistent analysis involves multiple reads (two or more) of the same row, and each time the information is changed by another transaction; thus, the term nonrepeatable read.

For example, an editor reads the same document twice, but between each reading the writer rewrites the document. When the editor reads the document for the second time, it has changed. The original read was not repeatable. This problem could be avoided if the writer could not change the document until the editor has finished reading it for the last time.

- **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 `SELECT` statements in the first transaction may return different results because the `INSERT` statement in the second transaction changes the data used by both.

```
--Transaction 1
BEGIN TRAN;
SELECT ID FROM dbo.employee
WHERE ID > 5 and ID < 10;
--The INSERT statement from the second transaction occurs here.
SELECT ID FROM dbo.employee
WHERE ID > 5 and ID < 10;
COMMIT;
```

```
--Transaction 2
BEGIN TRAN;
INSERT INTO dbo.employee
   SET name = 'New' WHERE ID = 5;
COMMIT;
```

- **Missing and double reads caused by row updates**

  - Missing a updated row or seeing an updated row multiple times

    Transactions that are running at the `READ UNCOMMITTED` level do not issue shared locks to prevent other transactions from modifying data read by the current transaction. Transactions that are running at the READ COMMITTED level do issue shared locks, but the row or page locks are released after the row is read. In either case, when you are scanning an index, if another user changes the index key column of the row during your read, the row might appear again if the key change moved the row to a position ahead of your scan. Similarly, the row might not appear if the key change moved the row to a position in the index that you had already read. To avoid this, use the `SERIALIZABLE` or `HOLDLOCK` hint, or row versioning. For more information, see Table Hints (Transact-SQL).

  - Missing one or more rows that were not the target of update

    When you are using `READ UNCOMMITTED`, if your query reads rows using an allocation order scan (using IAM pages), you might miss rows if another transaction is causing a page split. This cannot occur when you are using read committed because a table lock is held during a page split and does not happen if the table does not have a clustered index, because updates do not cause page splits.

#### 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.

- **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.

  SQL Server supports a range of concurrency control. Users specify the type of concurrency control by selecting transaction isolation levels for connections or concurrency options on cursors. These attributes can be defined using Transact-SQL statements, or through the properties and attributes of database application programming interfaces (APIs) such as ADO, ADO.NET, OLE DB, and ODBC.

#### 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. Isolation levels are described in terms of which concurrency side-effects, such as dirty reads or phantom reads, are allowed.

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.

> **IMPORTANT**
>
> Choosing a transaction isolation level does not affect the locks acquired to protect data modifications. A transaction always gets an exclusive lock on any data it modifies, and holds that lock until the transaction completes, regardless of the isolation level set for that transaction. For read operations, transaction isolation levels primarily define the level of protection from the effects of modifications made by other transactions.

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. Choosing the appropriate isolation level depends on balancing the data integrity requirements of the application against the overhead of each isolation level. The highest isolation level, serializable, guarantees that a transaction will retrieve exactly the same data every time it repeats a read operation, but it does this by performing a level of locking that is likely to impact other users in multi-user systems. The lowest isolation level, read uncommitted, may retrieve data that has been modified but not committed by other transactions. All of the concurrency side effects can happen in read uncommitted, but there is no read locking or versioning, so overhead is minimized.

The ISO standard defines the following isolation levels, all of which are supported by the SQL Server Database Engine:

| ISOLATION LEVEL | DEFINITION |
| --- | --- |
| Read uncommitted | The lowest isolation level where transactions are isolated only enough to ensure that physically corrupt data is not read. In this level, dirty reads are allowed, so one transaction may see not-yet-committed changes made by other transactions. |
| Read committed | Allows a transaction to read data previously read (not modified) by another transaction without waiting for the first transaction to complete. The SQL Server Database Engine keeps write locks (acquired on selected data) until the end of the transaction, but read locks are released as soon as the SELECT operation is performed. This is the SQL Server Database Engine default level. |
| Repeatable read | The SQL Server Database Engine keeps read and write locks that are acquired on selected data until the end of the transaction. However, because range-locks are not managed, phantom reads can occur. |
| Serializable | The highest level where transactions are completely isolated from one another. The SQL Server Database Engine keeps read and write locks acquired on selected data to be released at the end of the transaction. Range-locks are acquired when a SELECT operation uses a ranged WHERE clause, especially to avoid phantom reads.<br><br>**Note:** DDL operations and transactions on replicated tables may fail when serializable isolation level is requested. This is because replication queries use hints that may be incompatible with serializable isolation level. |

SQL Server also supports two additional transaction isolation levels that use row versioning. One is an implementation of read committed isolation, and one is a transaction isolation level, snapshot.

| ROW VERSIONING ISOLATION LEVEL | DEFINITION |
| --- | --- |
| 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. That is, the SQL Server Database Engine uses row versioning to present each statement with a transactionally consistent snapshot of the data as it existed at the start of the statement. 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.<br><br>When the `READ_COMMITTED_SNAPSHOT` database option is set OFF, which is the default setting, read committed isolation uses shared locks to prevent other transactions from modifying rows while the current transaction is running a read operation. The shared locks also block the statement from reading rows modified by other transactions until the other transaction is completed. Both implementations meet the ISO definition of read committed isolation. |

| ROW VERSIONING ISOLATION LEVEL | DEFINITION |
|---|---|
| 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. When reading rows modified by another transaction, they retrieve the version of the row that existed when the transaction started. You can only use Snapshot isolation against a database when the `ALLOW_SNAPSHOT_ISOLATION` database option is set ON. By default, this option is set OFF for user databases.<br><br>**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 following DDL statements are not permitted under snapshot isolation after a BEGIN TRANSACTION statement: ALTER TABLE, CREATE INDEX, CREATE XML INDEX, ALTER INDEX, DROP INDEX, DBCC REINDEX, ALTER PARTITION FUNCTION, ALTER PARTITION SCHEME, or any common language runtime (CLR) DDL statement. These statements are permitted when you are using snapshot isolation within implicit transactions. An implicit transaction, by definition, is a single statement that makes it possible to enforce the semantics of snapshot isolation, even with DDL statements. Violations of this principle can cause error 3961:<br>`Snapshot isolation transaction failed in database '%.*ls' because the object accessed by the statement has been modified by a DDL statement in another concurrent transaction since the start of this transaction. It is not allowed because the metadata is not versioned. A concurrent update to metadata could lead to inconsistency if mixed with snapshot isolation.` |

The following table shows the concurrency side effects enabled by the different isolation levels.

| 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 |

For more information about the specific types of locking or row versioning controlled by each transaction isolation level, see SET TRANSACTION ISOLATION LEVEL (Transact-SQL).

Transaction isolation levels can be set using Transact-SQL or through a database API.

Transact-SQL scripts use the SET TRANSACTION ISOLATION LEVEL statement.

### ADO

ADO applications set the `IsolationLevel` property of the **Connection** object to adXactReadUncommitted, adXactReadCommitted, adXactRepeatableRead, or adXactReadSerializable.

**ADO.NET**

ADO.NET applications using the `System.Data.SqlClient` managed namespace can call the `SqlConnection.BeginTransaction` method and set the *IsolationLevel* option to Unspecified, Chaos, ReadUncommitted, ReadCommitted, RepeatableRead, Serializable, and Snapshot.

**OLE DB**

When starting a transaction, applications using OLE DB call `ITransactionLocal::StartTransaction` with *isoLevel* set to ISOLATIONLEVEL_READUNCOMMITTED, ISOLATIONLEVEL_READCOMMITTED, ISOLATIONLEVEL_REPEATABLEREAD, ISOLATIONLEVEL_SNAPSHOT, or ISOLATIONLEVEL_SERIALIZABLE.

When specifying the transaction isolation level in autocommit mode, OLE DB applications can set the DBPROPSET_SESSION property DBPROP_SESS_AUTOCOMMITISOLEVELS to DBPROPVAL_TI_CHAOS, DBPROPVAL_TI_READUNCOMMITTED, DBPROPVAL_TI_BROWSE, DBPROPVAL_TI_CURSORSTABILITY, DBPROPVAL_TI_READCOMMITTED, DBPROPVAL_TI_REPEATABLEREAD, DBPROPVAL_TI_SERIALIZABLE, DBPROPVAL_TI_ISOLATED, or DBPROPVAL_TI_SNAPSHOT.

**ODBC**

ODBC applications call `SQLSetConnectAttr` with *Attribute* set to SQL_ATTR_TXN_ISOLATION and *ValuePtr* set to SQL_TXN_READ_UNCOMMITTED, SQL_TXN_READ_COMMITTED, SQL_TXN_REPEATABLE_READ, or SQL_TXN_SERIALIZABLE.

For snapshot transactions, applications call `SQLSetConnectAttr` with Attribute set to SQL_COPT_SS_TXN_ISOLATION and ValuePtr set to SQL_TXN_SS_SNAPSHOT. A snapshot transaction can be retrieved using either SQL_COPT_SS_TXN_ISOLATION or SQL_ATTR_TXN_ISOLATION.

# 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. No transaction can be granted a lock that would conflict with the mode of a lock already granted on that data to another transaction. If a transaction requests a lock mode that conflicts with a lock that has already been granted on the same data, the instance of the SQL Server Database Engine will pause the requesting transaction until the first lock is released.

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. When an instance of the SQL Server Database Engine processes a Transact-SQL statement, the SQL Server Database Engine query processor determines which resources are to be accessed. The query processor determines what types of locks are required to protect each resource based on the type of access and the transaction isolation level setting. The query processor then requests the appropriate locks from the lock manager. The lock manager grants the locks if there are no conflicting locks held by other transactions.

**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.

The SQL Server Database Engine often has to acquire locks at multiple levels of granularity to fully protect a resource. This group of locks at multiple levels of granularity is called a lock hierarchy. For example, to fully protect a read of an index, an instance of the SQL Server Database Engine may have to acquire share locks on rows and intent share locks on the pages and table.

The following table shows the resources that the SQL Server Database Engine can lock.

| 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. |

**NOTE**

HoBT and TABLE locks can be affected by the LOCK_ESCALATION option of ALTER TABLE.

### Lock Modes

The SQL Server Database Engine locks resources using different lock modes that determine how the resources can be accessed by concurrent transactions.

The following table shows the resource lock modes that the SQL Server Database Engine uses.

| LOCK MODE | DESCRIPTION |
| --- | --- |

| 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`, or `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.

Data modification statements, such as INSERT, UPDATE, and DELETE combine both modification and read

operations. The statement first performs read operations to acquire data before performing the required modification operations. Data modification statements, therefore, typically request both shared locks and exclusive locks. For example, an UPDATE statement might modify rows in one table based on a join with another table. In this case, the UPDATE statement requests shared locks on the rows read in the join table in addition to requesting exclusive locks on the updated rows.

#### 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. |

| LOCK MODE | DESCRIPTION |
| --- | --- |
| 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. However, concurrent DDL operations, and concurrent DML operations that acquire Sch-M locks, cannot be performed on the table.

#### 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.

| | NL | SCH-S | SCH-M | S | U | X | IS | IU | IX | SIU | SIX | UIX | BU | RS-S | RS-U | RI-N | RI-S | RI-U | RI-X | RX-S | RX-U | RX-X |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NL | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N |
| SCH-S | N | N | C | N | N | N | N | N | N | N | N | N | N | I | I | I | I | I | I | I | I | I |
| SCH-M | N | C | C | C | C | C | C | C | C | C | C | C | C | I | I | I | I | I | I | I | I | I |
| S | N | N | C | N | N | C | N | N | C | N | C | C | C | N | N | N | N | N | C | N | N | C |
| U | N | N | C | N | C | C | N | C | C | C | C | C | C | N | C | N | N | C | C | N | C | C |
| X | N | N | C | C | C | C | C | C | C | C | C | C | C | C | C | N | C | C | C | C | C | C |
| IS | N | N | C | N | N | C | N | N | N | N | N | N | C | I | I | I | I | I | I | I | I | I |
| IU | N | N | C | N | C | C | N | N | N | N | N | C | C | I | I | I | I | I | I | I | I | I |
| IX | N | N | C | C | C | C | N | N | N | C | C | C | C | I | I | I | I | I | I | I | I | I |
| SIU | N | N | C | N | C | C | N | N | C | N | C | C | C | I | I | I | I | I | I | I | I | I |
| SIX | N | N | C | C | C | C | N | N | C | C | C | C | C | I | I | I | I | I | I | I | I | I |
| UIX | N | N | C | C | C | C | N | C | C | C | C | C | C | I | I | I | I | I | I | I | I | I |
| BU | N | N | C | C | C | C | C | C | C | C | C | C | N | I | I | I | I | I | I | I | I | I |
| RS-S | N | I | I | N | N | C | I | I | I | I | I | I | I | N | N | C | C | C | C | C | C | C |
| RS-U | N | I | I | N | C | C | I | I | I | I | I | I | I | N | C | C | C | C | C | C | C | C |
| RI-N | N | I | I | N | N | N | I | I | I | I | I | I | I | C | C | N | N | N | N | C | C | C |
| RI-S | N | I | I | N | N | C | I | I | I | I | I | I | I | C | C | N | N | N | C | C | C | C |
| RI-U | N | I | I | N | C | C | I | I | I | I | I | I | I | C | C | N | N | C | C | C | C | C |
| RI-X | N | I | I | C | C | C | I | I | I | I | I | I | I | C | C | N | C | C | C | C | C | C |
| RX-S | N | I | I | N | N | C | I | I | I | I | I | I | I | C | C | C | C | C | C | C | C | C |
| RX-U | N | I | I | N | C | C | I | I | I | I | I | I | I | C | C | C | C | C | C | C | C | C |
| RX-X | N | I | I | C | C | C | I | I | I | I | I | I | I | C | C | C | C | C | C | C | C | C |

**Key**

| | | | |
|---|---|---|---|
| N | No Conflict | SIU | Share with Intent Update |
| I | Illegal | SIX | Shared with Intent Exclusive |
| C | Conflict | UIX | Update with Intent Exclusive |
| | | BU | Bulk Update |
| NL | No Lock | RS-S | Shared Range-Shared |
| SCH-S | Schema Stability Locks | RS-U | Shared Range-Update |
| SCH-M | Schema Modification Locks | RI-N | Insert Range-Null |
| S | Shared | RI-S | Insert Range-Shared |
| U | Update | RI-U | Insert Range-Update |
| X | Exclusive | RI-X | Insert Range-Exclusive |
| IS | Intent Shared | RX-S | Exclusive Range-Shared |
| IU | Intent Update | RX-U | Exclusive Range-Update |
| IX | Intent Exclusive | RX-X | Exclusive Range-Exclusive |

## Key-Range Locking

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. The serializable isolation level requires that any query executed during a transaction must obtain the same set of rows every time it is executed during the transaction. A key range lock protects this requirement by preventing other transactions from inserting new rows whose keys would fall in the range of keys read by the serializable transaction.

Key-range locking prevents phantom reads. By protecting the ranges of keys between rows, it also prevents phantom insertions into a set of records accessed by a transaction.

A key-range lock is placed on an index, specifying a beginning and ending key value. This lock blocks any attempt to insert, update, or delete any row with a key value that falls in the range because those operations would first have to acquire a lock on the index. For example, a serializable transaction could issue a SELECT statement that reads all rows whose key values are between 'AAA' and 'CZZ'. A key-range lock on the key values in the range from 'AAA' to 'CZZ' prevents other transactions from inserting rows with key values anywhere in that range, such as 'ADG', 'BBD', or 'CAL'.

### Key-Range Lock Modes

Key-range locks include both a range and a row component specified in range-row format:

- Range represents the lock mode protecting the range between two consecutive index entries.
- Row represents the lock mode protecting the index entry.
- Mode represents the combined lock mode used. Key-range lock modes consist of two parts. The first represents the type of lock used to lock the index range (Range*T*) and the second represents the lock type used to lock a specific key (*K*). The two parts are connected with a hyphen (-), such as Range*T*-*K*.

| RANGE | ROW | MODE | DESCRIPTION |
|---|---|---|---|
| RangeS | S | RangeS-S | Shared range, shared resource lock; serializable range scan. |
| RangeS | U | RangeS-U | Shared range, update resource lock; serializable update scan. |

| RANGE | ROW | MODE | DESCRIPTION |
| --- | --- | --- | --- |
| RangeI | Null | RangeI-N | Insert range, null resource lock; used to test ranges before inserting a new key into an index. |
| RangeX | X | RangeX-X | Exclusive range, exclusive resource lock; used when updating a key in a range. |

**NOTE**

The internal Null lock mode is compatible with all other lock modes.

Key-range lock modes have a compatibility matrix that shows which locks are compatible with other locks obtained on overlapping keys and ranges.

| | EXISTING GRANTED MODE | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Requested mode** | **S** | **U** | **X** | **RangeS-S** | **RangeS-U** | **RangeI-N** | **RangeX-X** |
| **Shared (S)** | Yes | Yes | No | Yes | Yes | Yes | No |
| **Update (U)** | Yes | No | No | Yes | No | Yes | No |
| **Exclusive (X)** | No | No | No | No | No | Yes | No |
| **RangeS-S** | Yes | Yes | No | Yes | Yes | No | No |
| **RangeS-U** | Yes | No | No | Yes | No | No | No |
| **RangeI-N** | Yes | Yes | Yes | No | No | Yes | No |
| **RangeX-X** | No | No | No | No | No | No | No |

#### Conversion Locks

Conversion locks are created when a key-range lock overlaps another lock.

| LOCK 1 | LOCK 2 | CONVERSION LOCK |
| --- | --- | --- |
| S | RangeI-N | RangeI-S |
| U | RangeI-N | RangeI-U |
| X | RangeI-N | RangeI-X |
| RangeI-N | RangeS-S | RangeX-S |
| RangeI-N | RangeS-U | RangeX-U |

Conversion locks can be observed for a short period of time under different complex circumstances, sometimes while running concurrent processes.

#### Serializable Range Scan, Singleton Fetch, Delete, and Insert

Key-range locking ensures that the following operations are serializable:

- Range scan query
- Singleton fetch of nonexistent row
- Delete operation
- Insert operation

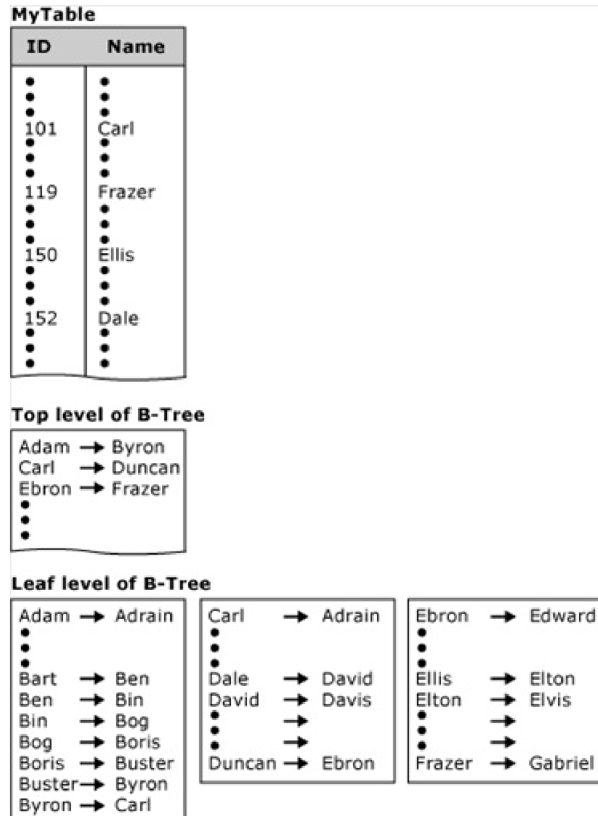
  Before key-range locking can occur, the following conditions must be satisfied:

- The transaction-isolation level must be set to SERIALIZABLE.

- The query processor must use an index to implement the range filter predicate. For example, the WHERE clause in a SELECT statement could establish a range condition with this predicate: ColumnX BETWEEN N'AAA' AND N'CZZ'. A key-range lock can only be acquired if **ColumnX** is covered by an index key.

#### Examples

The following table and index are used as a basis for the key-range locking examples that follow.




##### Range Scan Query

To ensure a range scan query is serializable, the same query should return the same results each time it is executed within the same transaction. New rows must not be inserted within the range scan query by other transactions; otherwise, these become phantom inserts. For example, the following query uses the table and index in the previous illustration:

```
SELECT name
FROM mytable
WHERE name BETWEEN 'A' AND 'C';
```

Key-range locks are placed on the index entries corresponding to the range of data rows where the name is between the values Adam and Dale, preventing new rows qualifying in the previous query from being added or

deleted. Although the first name in this range is Adam, the RangeS-S mode key-range lock on this index entry ensures that no new names beginning with the letter A can be added before Adam, such as Abigail. Similarly, the RangeS-S key-range lock on the index entry for Dale ensures that no new names beginning with the letter C can be added after Carlos, such as Clive.

> **NOTE**
>
> The number of RangeS-S locks held is *n*+1, where *n* is the number of rows that satisfy the query.

###### Singleton Fetch of Nonexistent Data

If a query within a transaction attempts to select a row that does not exist, issuing the query at a later point within the same transaction has to return the same result. No other transaction can be allowed to insert that nonexistent row. For example, given this query:

```
SELECT name
FROM mytable
WHERE name = 'Bill';
```

A key-range lock is placed on the index entry corresponding to the name range from `Ben` to `Bing` because the name `Bill` would be inserted between these two adjacent index entries. The RangeS-S mode key-range lock is placed on the index entry `Bing`. This prevents any other transaction from inserting values, such as `Bill`, between the index entries `Ben` and `Bing`.

###### Delete Operation

When deleting a value within a transaction, the range the value falls into does not have to be locked for the duration of the transaction performing the delete operation. Locking the deleted key value until the end of the transaction is sufficient to maintain serializability. For example, given this DELETE statement:

```
DELETE mytable
WHERE name = 'Bob';
```

An exclusive (X) lock is placed on the index entry corresponding to the name `Bob`. Other transactions can insert or delete values before or after the deleted value `Bob`. However, any transaction that attempts to read, insert, or delete the value `Bob` will be blocked until the deleting transaction either commits or rolls back.

Range delete can be executed using three basic lock modes: row, page, or table lock. The row, page, or table locking strategy is decided by query optimizer or can be specified by the user through optimizer hints such as ROWLOCK, PAGLOCK, or TABLOCK. When PAGLOCK or TABLOCK is used, the SQL Server Database Engine immediately deallocates an index page if all rows are deleted from this page. In contrast, when ROWLOCK is used, all deleted rows are marked only as deleted; they are removed from the index page later using a background task.

###### Insert Operation

When inserting a value within a transaction, the range the value falls into does not have to be locked for the duration of the transaction performing the insert operation. Locking the inserted key value until the end of the transaction is sufficient to maintain serializability. For example, given this INSERT statement:

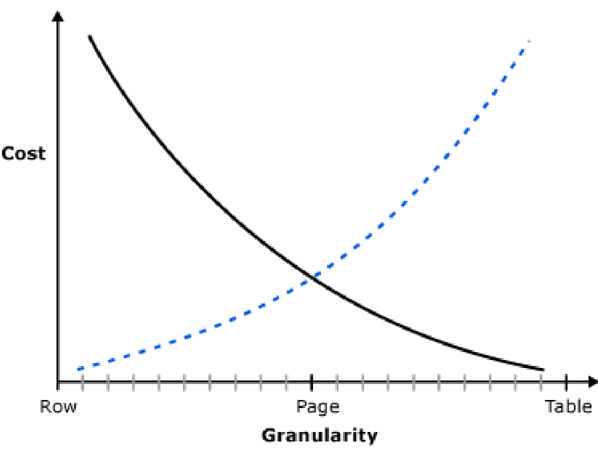
```
INSERT mytable VALUES ('Dan');
```

The RangeI-N mode key-range lock is placed on the index entry corresponding to the name David to test the range. If the lock is granted, `Dan` is inserted and an exclusive (X) lock is placed on the value `Dan`. The RangeI-N mode key-range lock is necessary only to test the range and is not held for the duration of the transaction performing the insert operation. Other transactions can insert or delete values before or after the inserted value `Dan`. However, any transaction attempting to read, insert, or delete the value `Dan` will be locked until the inserting

transaction either commits or rolls back.

### 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.

  In SQL Server 2008 and later versions, the behavior of lock escalation has changed with the introduction of the `LOCK_ESCALATION` option. For more information, see the `LOCK_ESCALATION` option of ALTER TABLE.

### 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.

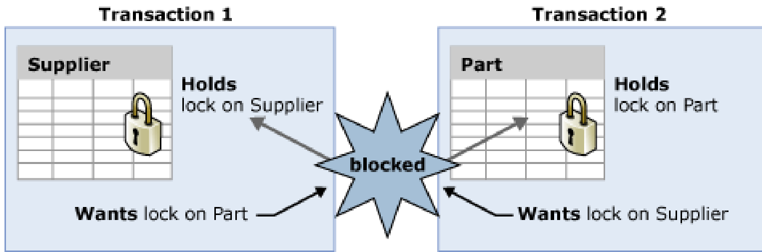
  Both transactions in a deadlock will wait forever unless the deadlock is broken by an external process. The

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.



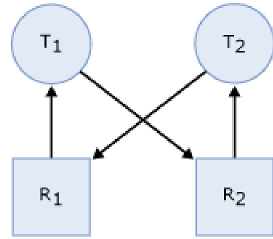

In the illustration, transaction T1 has a dependency on transaction T2 for the **Part** table lock resource. Similarly, transaction T2 has a dependency on transaction T1 for the **Supplier** table lock resource. Because these dependencies form a cycle, there is a deadlock between transactions T1 and T2.

Deadlocks can also occur when a table is partitioned and the `LOCK_ESCALATION` setting of `ALTER TABLE` is set to AUTO. When `LOCK_ESCALATION` is 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. The following graph presents a high level view of a deadlock state where:

- 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.




The SQL Server Database Engine automatically detects deadlock cycles within SQL Server. The SQL Server Database Engine chooses one of the sessions as a deadlock victim and the current transaction is terminated with an error to break the deadlock.

###### 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).

  - **User resource**. When a thread is waiting for a resource that is potentially controlled by a user application, the resource is considered to be 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.

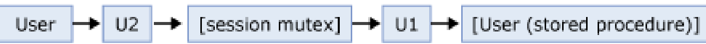
  - **Transaction mutex**. All tasks running in one transaction are interleaved, meaning that only one task can run under the transaction at a given time. Before the task can run, it must have exclusive access to the transaction mutex.

  In order for a task to run under MARS, it must acquire the session mutex. If the task is running under a transaction, it must then acquire the transaction mutex. This guarantees that only one task is active

at one time in a given session and a given transaction. Once the required mutexes have been acquired, the task can execute. When the task finishes, or yields in the middle of the request, it will first release transaction mutex followed by the session mutex in reverse order of acquisition. However, deadlocks can occur with these resources. In the following code example, two tasks, user request U1 and user request U2, are running in the same session.

```
U1:    Rs1=Command1.Execute("insert sometable EXEC usp_someproc");
U2:    Rs2=Command2.Execute("select colA from sometable");
```

The stored procedure executing from user request U1 has acquired the session mutex. If the stored procedure takes a long time to execute, it is assumed by the SQL Server Database Engine that the stored procedure is waiting for input from the user. User request U2 is waiting for the session mutex while the user is waiting for the result set from U2, and U1 is waiting for a user resource. This is deadlock state logically illustrated as:




###### 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.
- If a deadlock has just been detected, it is assumed that the next threads that must wait for a lock are entering the deadlock cycle. The first couple of lock waits after a deadlock has been detected will immediately trigger a deadlock search rather than wait for the next deadlock detection interval. For example, if the current interval is 5 seconds, and a deadlock was just detected, the next lock wait will kick off the deadlock detector immediately. If this lock wait is part of a deadlock, it will be detected right away rather than during next deadlock search.

  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.

When working with CLR, the deadlock monitor automatically detects deadlock for synchronization resources (monitors, reader/writer lock and thread join) accessed inside managed procedures. However, the deadlock is resolved by throwing an exception in the procedure that was selected to be the deadlock victim. It is important to understand that the exception does not automatically release resources currently owned by the victim; the resources must be explicitly released. Consistent with exception behavior, the exception used to identify a deadlock victim can be caught and dismissed.

#### Deadlock Information Tools

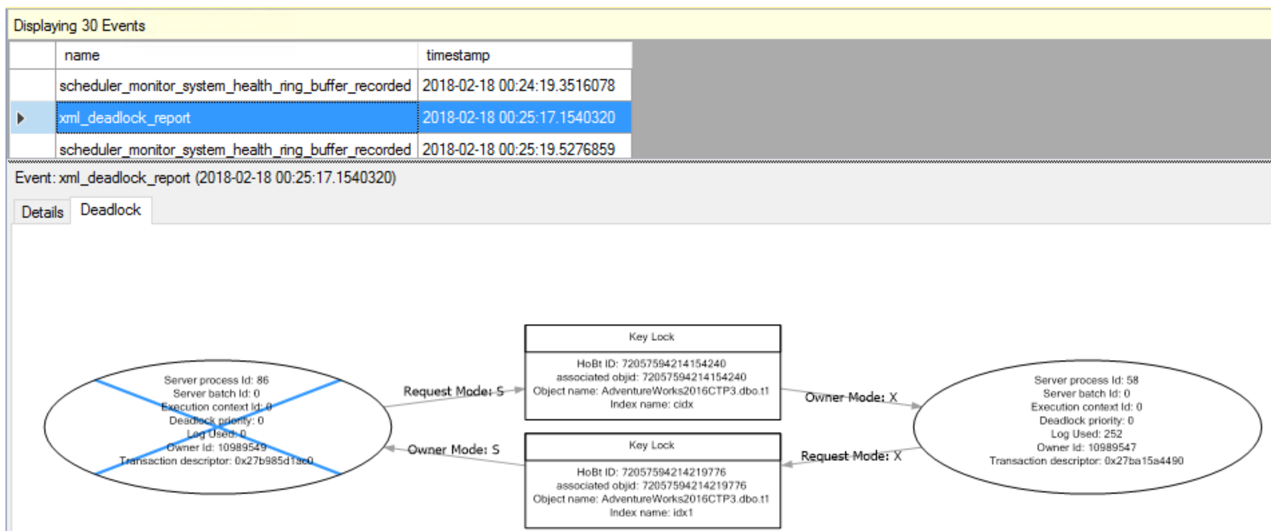
To view deadlock information, the SQL Server Database Engine provides monitoring tools in the form of the the system_health xEvent session, two trace flags, and the deadlock graph event in SQL Profiler.

##### Deadlock in system_health session

Starting with SQL Server 2012 (11.x), when deadlocks occur, the system_health session captures all `xml_deadlock_report` xEvents. The system_health session is enabled by default. The deadlock graph captured typically has three distinct nodes:

- **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, as seen in the following example:




The following query can view all deadlock events captured by the system_health session ring buffer.

```
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
```

Here is the result set.

| | Date | Event_Data |
|---|---|---|
| 1 | 2018-02-18 08:26:24.697 | <event name="xml_deadlock_report" package="sqlse... |
| 2 | 2018-02-18 08:25:17.153 | <event name="xml_deadlock_report" package="sqlse... |

The following example shows the output, after clicking on the first link of the result above:

```
<event name="xml_deadlock_report" package="sqlserver" timestamp="2018-02-18T08:26:24.698Z">
  <data name="xml_report">
    <type name="xml" package="package0" />
    <value>
      <deadlock>
        <victim-list>
          <victimProcess id="process27b9b0b9848" />
        </victim-list>
        <process-list>
          <process id="process27b9b0b9848" taskpriority="0" logused="0" waitresource="KEY: 5:72057594214350848
(1a39e6095155)" waittime="1631" ownerId="11088595" transactionname="SELECT" lasttranstarted="2018-02-
18T00:26:23.073" XDES="0x27b9f79fac0" lockMode="S" schedulerid="9" kpid="15336" status="suspended" spid="62"
sbid="0" ecid="0" priority="0" trancount="0" lastbatchstarted="2018-02-18T00:26:22.893"
lastbatchcompleted="2018-02-18T00:26:22.890" lastattention="1900-01-01T00:00:00.890" clientapp="SQLCMD"
hostname="ContosoServer" hostpid="7908" loginname="CONTOSO\user" isolationlevel="read committed (2)"
xactid="11088595" currentdb="5" lockTimeout="4294967295" clientoption1="538968096" clientoption2="128056">
            <executionStack>
              <frame procname="AdventureWorks2016CTP3.dbo.p1" line="3" stmtstart="78" stmtend="180"
sqlhandle="0x0300050020766505ca3e07008ba8000001000000000000000000000000000000000000000000000000000000">
SELECT c2, c3 FROM t1 WHERE c2 BETWEEN @p1 AND @p1+    </frame>
              <frame procname="adhoc" line="4" stmtstart="82" stmtend="98"
sqlhandle="0x020000006263ec01ebb919c335024a072a2699958d3fcce60000000000000000000000000000000000000000">
unknown    </frame>
            </executionStack>
            <inputbuf>
SET NOCOUNT ON
WHILE (1=1)
BEGIN
    EXEC p1 4
END
   </inputbuf>
          </process>
          <process id="process27b9ee33c28" taskpriority="0" logused="252" waitresource="KEY:
5:72057594214416384 (e5b3d7e750dd)" waittime="1631" ownerId="11088593" transactionname="UPDATE"
lasttranstarted="2018-02-18T00:26:23.073" XDES="0x27ba15a4490" lockMode="X" schedulerid="6" kpid="5584"
status="suspended" spid="58" sbid="0" ecid="0" priority="0" trancount="2" lastbatchstarted="2018-02-
18T00:26:22.890" lastbatchcompleted="2018-02-18T00:26:22.890" lastattention="1900-01-01T00:00:00.890"
clientapp="SQLCMD" hostname="ContosoServer" hostpid="15316" loginname="CONTOSO\user" isolationlevel="read
committed (2)" xactid="11088593" currentdb="5" lockTimeout="4294967295" clientoption1="538968096"
clientoption2="128056">
            <executionStack>
              <frame procname="AdventureWorks2016CTP3.dbo.p2" line="3" stmtstart="76" stmtend="150"
sqlhandle="0x03000500599a5906ce3e07008ba8000001000000000000000000000000000000000000000000000000000000">
UPDATE t1 SET c2 = c2+1 WHERE c1 = @p    </frame>
              <frame procname="adhoc" line="4" stmtstart="82" stmtend="98"
sqlhandle="0x0200000008fe521e5fb1099410048c5743ff7da04b2047b00000000000000000000000000000000000000000">
unknown    </frame>
            </executionStack>
            <inputbuf>
SET NOCOUNT ON
WHILE (1=1)
BEGIN
    EXEC p2 4
END
   </inputbuf>
          </process>
        </process-list>
        <resource-list>
          <keylock hobtid="72057594214350848" dbid="5" objectname="AdventureWorks2016CTP3.dbo.t1"
```

```
          <keylock hobtid="72057594214350848" dbid="5" objectname="AdventureWorks2016CTP3.dbo.t1"
indexname="cidx" id="lock27b9dd26a00" mode="X" associatedObjectId="72057594214350848">
            <owner-list>
              <owner id="process27b9ee33c28" mode="X" />
            </owner-list>
            <waiter-list>
              <waiter id="process27b9b0b9848" mode="S" requestType="wait" />
            </waiter-list>
          </keylock>
          <keylock hobtid="72057594214416384" dbid="5" objectname="AdventureWorks2016CTP3.dbo.t1"
indexname="idx1" id="lock27afa392600" mode="S" associatedObjectId="72057594214416384">
            <owner-list>
              <owner id="process27b9b0b9848" mode="S" />
            </owner-list>
            <waiter-list>
              <waiter id="process27b9ee33c28" mode="X" requestType="wait" />
            </waiter-list>
          </keylock>
        </resource-list>
      </deadlock>
    </value>
  </data>
</event>
```

For more information, see Use the system_health Session

#### Trace Flag 1204 and Trace Flag 1222

When deadlocks occur, trace flag 1204 and trace flag 1222 return information that is captured in the SQL Server error log. Trace flag 1204 reports deadlock information formatted by each node involved in the deadlock. Trace flag 1222 formats deadlock information, first by processes and then by resources. It is possible to enable both trace flags to obtain two representations of the same deadlock event.

In addition to defining the properties of trace flag 1204 and 1222, the following table also shows the similarities and differences.

| PROPERTY | TRACE FLAG 1204 AND TRACE FLAG 1222 | TRACE FLAG 1204 ONLY | TRACE FLAG 1222 ONLY |
| --- | --- | --- | --- |
| Output format | Output is captured in the SQL Server error log. | Focused on the nodes involved in the deadlock. Each node has a dedicated section, and the final section describes the deadlock victim. | Returns information in an XML-like format that does not conform to an XML Schema Definition (XSD) schema. The format has three major sections. The first section declares the deadlock victim. The second section describes each process involved in the deadlock. The third section describes the resources that are synonymous with nodes in trace flag 1204. |
| Identifying attributes | **SPID:<x> ECID:<x>.** Identifies the system process ID thread in cases of parallel processes. The entry `SPID:<x> ECID:0`, where <x> is replaced by the SPID value, represents the main thread. The entry `SPID:<x> ECID:<y>`, where <x> is replaced by the SPID value and <y> is greater than 0, represents the sub- | **Node**. Represents the entry number in the deadlock chain.<br><br>**Lists**. The lock owner can be part of these lists:<br><br>**Grant List**. Enumerates the current owners of the resource.<br><br>**Convert List**. Enumerates | **deadlock victim**. Represents the physical memory address of the task (see sys.dm_os_tasks (Transact-SQL)) that was selected as a deadlock victim. It may be 0 (zero) in the case of an unresolved deadlock. A task that is rolling back cannot be chosen as a deadlock victim. |

| PROPERTY | TRACE FLAG 1204 AND TRACE FLAG 1222 | TRACE FLAG 1204 ONLY | TRACE FLAG 1222 ONLY |
| --- | --- | --- | --- |
| | than 0, represents the sub-threads for the same SPID.<br><br>**BatchID** (**sbid** for trace flag 1222). Identifies the batch from which code execution is requesting or holding a lock. When Multiple Active Result Sets (MARS) is disabled, the BatchID value is 0. When MARS is enabled, the value for active batches is 1 to *n*. If there are no active batches in the session, BatchID is 0.<br><br>**Mode**. Specifies the type of lock for a particular resource that is requested, granted, or waited on by a thread. Mode can be IS (Intent Shared), S (Shared), U (Update), IX (Intent Exclusive), SIX (Shared with Intent Exclusive), and X (Exclusive).<br><br>**Line #** (**line** for trace flag 1222). Lists the line number in the current batch of statements that was being executed when the deadlock occurred.<br><br>**Input Buf** (**inputbuf** for trace flag 1222). Lists all the statements in the current batch. | the current owners that are trying to convert their locks to a higher level.<br><br>**Wait List**. Enumerates current new lock requests for the resource.<br><br>**Statement Type**. Describes the type of DML statement (SELECT, INSERT, UPDATE, or DELETE) on which the threads have permissions.<br><br>**Victim Resource Owner**. Specifies the participating thread that SQL Server chooses as the victim to break the deadlock cycle. The chosen thread and all existing sub-threads are terminated.<br><br>**Next Branch**. Represents the two or more sub-threads from the same SPID that are involved in the deadlock cycle. | **executionstack**. Represents Transact-SQL code that is being executed at the time the deadlock occurs.<br><br>**priority**. Represents deadlock priority. In certain cases, the SQL Server Database Engine may opt to alter the deadlock priority for a short duration to achieve better concurrency.<br><br>**logused**. Log space used by the task.<br><br>**owner id**. The ID of the transaction that has control of the request.<br><br>**status**. State of the task. It is one of the following values:<br><br>>> **pending**. Waiting for a worker thread.<br><br>>> **runnable**. Ready to run but waiting for a quantum.<br><br>>> **running**. Currently running on the scheduler.<br><br>>> **suspended**. Execution is suspended.<br><br>>> **done**. Task has completed.<br><br>>> **spinloop**. Waiting for a spinlock to become free.<br><br>**waitresource**. The resource needed by the task.<br><br>**waittime**. Time in milliseconds waiting for the resource.<br><br>**schedulerid**. Scheduler associated with this task. See sys.dm_os_schedulers (Transact-SQL).<br><br>**hostname**. The name of the workstation.<br><br>**isolationlevel**. The current transaction isolation level.<br><br>**Xactid**. The ID of the transaction that has control of the request.<br><br>**currentdb**. The ID of the database. |

| PROPERTY | TRACE FLAG 1204 AND TRACE FLAG 1222 | TRACE FLAG 1204 ONLY | TRACE FLAG 1222 ONLY |
| --- | --- | --- | --- |
| | | | **lastbatchstarted**. The last time a client process started batch execution.<br><br>**lastbatchcompleted**. The last time a client process completed batch execution.<br><br>**clientoption1 and clientoption2**. Set options on this client connection. This is a bitmask that includes information about options usually controlled by SET statements such as SET NOCOUNT and SET XACTABORT.<br><br>**associatedObjectId**. Represents the HoBT (heap or b-tree) ID. |
| Resource attributes | **RID**. Identifies the single row within a table on which a lock is held or requested. RID is represented as RID: *db_id:file_id:page_no:row_no*. For example, `RID: 6:1:20789:0`.<br><br>**OBJECT**. Identifies the table on which a lock is held or requested. OBJECT is represented as OBJECT: *db_id:object_id*. For example, `TAB: 6:2009058193`.<br><br>**KEY**. Identifies the key range within an index on which a lock is held or requested. KEY is represented as KEY: *db_id:hobt_id* (*index key hash value*). For example, `KEY: 6:72057594057457664 (350007a4d329)`.<br><br>**PAG**. Identifies the page resource on which a lock is held or requested. PAG is represented as PAG: *db_id:file_id:page_no*. For example, `PAG: 6:1:20789`.<br><br>**EXT**. Identifies the extent structure. EXT is represented as EXT: *db_id:file_id:extent_no*. For example, `EXT: 6:1:9`.<br><br>**DB**. Identifies the database lock. **DB is represented in one of the following ways:** | None exclusive to this trace flag. | None exclusive to this trace flag. |

| PROPERTY | TRACE FLAG 1204 AND TRACE FLAG 1222 | TRACE FLAG 1204 ONLY | TRACE FLAG 1222 ONLY |
|---|---|---|---|
| | DB: *db_id*<br>DB: *db_id*[BULK-OP-DB], which identifies the database lock taken by the backup database.<br><br>DB: *db_id*[BULK-OP-LOG], which identifies the lock taken by the backup log for that particular database.<br><br>**APP**. Identifies the lock taken by an application resource. APP is represented as APP: *lock_resource*. For example, `APP: Formf370f478`.<br><br>**METADATA**. Represents metadata resources involved in a deadlock. Because METADATA has many subresources, the value returned depends upon the subresource that has deadlocked. For example, METADATA.USER_TYPE returns `user_type_id =` <*integer_value*>. For more information about METADATA resources and subresources, see sys.dm_tran_locks (Transact-SQL).<br><br>**HOBT**. Represents a heap or b-tree involved in a deadlock. | | |

###### Trace Flag 1204 Example

The following example shows the output when trace flag 1204 is turned on. In this case, the table in Node 1 is a heap with no indexes, and the table in Node 2 is a heap with a nonclustered index. The index key in Node 2 is being updated when the deadlock occurs.

```
Deadlock encountered .... Printing deadlock information
Wait-for graph

Node:1

RID: 6:1:20789:0               CleanCnt:3 Mode:X Flags: 0x2
 Grant List 0:
   Owner:0x0315D6A0 Mode: X
     Flg:0x0 Ref:0 Life:02000000 SPID:55 ECID:0 XactLockInfo: 0x04D9E27C
   SPID: 55 ECID: 0 Statement Type: UPDATE Line #: 6
   Input Buf: Language Event:
BEGIN TRANSACTION
   EXEC usp_p2
 Requested By:
   ResType:LockOwner Stype:'OR'Xdes:0x03A3DAD0
     Mode: U SPID:54 BatchID:0 ECID:0 TaskProxy:(0x04976374) Value:0x315d200 Cost:(0/868)

Node:2

KEY: 6:72057594057457664 (350007a4d329) CleanCnt:2 Mode:X Flags: 0x0
 Grant List 0:
   Owner:0x0315D140 Mode: X
     Flg:0x0 Ref:0 Life:02000000 SPID:54 ECID:0 XactLockInfo: 0x03A3DAF4
   SPID: 54 ECID: 0 Statement Type: UPDATE Line #: 6
   Input Buf: Language Event:
     BEGIN TRANSACTION
       EXEC usp_p1
 Requested By:
   ResType:LockOwner Stype:'OR'Xdes:0x04D9E258
     Mode: U SPID:55 BatchID:0 ECID:0 TaskProxy:(0x0475E374) Value:0x315d4a0 Cost:(0/380)

Victim Resource Owner:
 ResType:LockOwner Stype:'OR'Xdes:0x04D9E258
     Mode: U SPID:55 BatchID:0 ECID:0 TaskProxy:(0x0475E374) Value:0x315d4a0 Cost:(0/380)
```

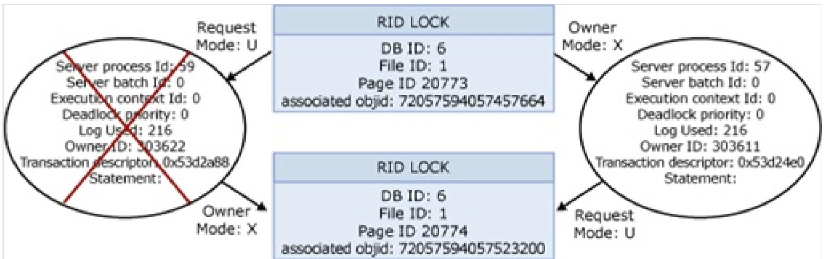
###### Trace Flag 1222 Example

The following example shows the output when trace flag 1222 is turned on. In this case, one table is a heap with no indexes, and the other table is a heap with a nonclustered index. In the second table, the index key is being updated when the deadlock occurs.

```
deadlock-list
 deadlock victim=process689978
  process-list
   process id=process6891f8 taskpriority=0 logused=868
   waitresource=RID: 6:1:20789:0 waittime=1359 ownerId=310444
   transactionname=user_transaction
   lasttranstarted=2005-09-05T11:22:42.733 XDES=0x3a3dad0
   lockMode=U schedulerid=1 kpid=1952 status=suspended spid=54
   sbid=0 ecid=0 priority=0 transcount=2
   lastbatchstarted=2005-09-05T11:22:42.733
   lastbatchcompleted=2005-09-05T11:22:42.733
   clientapp=Microsoft SQL Server Management Studio - Query
   hostname=TEST_SERVER hostpid=2216 loginname=DOMAIN\user
   isolationlevel=read committed (2) xactid=310444 currentdb=6
   lockTimeout=4294967295 clientoption1=671090784 clientoption2=390200
    executionStack
     frame procname=AdventureWorks2016.dbo.usp_p1 line=6 stmtstart=202
     sqlhandle=0x0300060013e6446b027cbb00c69600000100000000000000
     UPDATE T2 SET COL1 = 3 WHERE COL1 = 1;
     frame procname=adhoc line=3 stmtstart=44
     sqlhandle=0x01000600856aa70f503b8104000000000000000000000000
     EXEC usp_p1
    inputbuf
      BEGIN TRANSACTION
       EXEC usp_p1
   process id=process689978 taskpriority=0 logused=380
   waitresource=KEY: 6:72057594057457664 (350007a4d329)
   waittime=5015 ownerId=310462 transactionname=user_transaction
   lasttranstarted=2005-09-05T11:22:44.077 XDES=0x4d9e258 lockMode=U
   schedulerid=1 kpid=3024 status=suspended spid=55 sbid=0 ecid=0
   priority=0 transcount=2 lastbatchstarted=2005-09-05T11:22:44.077
   lastbatchcompleted=2005-09-05T11:22:44.077
   clientapp=Microsoft SQL Server Management Studio - Query
   hostname=TEST_SERVER hostpid=2216 loginname=DOMAIN\user
   isolationlevel=read committed (2) xactid=310462 currentdb=6
   lockTimeout=4294967295 clientoption1=671090784 clientoption2=390200
    executionStack
     frame procname=AdventureWorks2016.dbo.usp_p2 line=6 stmtstart=200
     sqlhandle=0x030006004c0a396c027cbb00c69600000100000000000000
     UPDATE T1 SET COL1 = 4 WHERE COL1 = 1;
     frame procname=adhoc line=3 stmtstart=44
     sqlhandle=0x01000600d688e709b85f8904000000000000000000000000
     EXEC usp_p2
    inputbuf
      BEGIN TRANSACTION
        EXEC usp_p2
  resource-list
   ridlock fileid=1 pageid=20789 dbid=6 objectname=AdventureWorks2016.dbo.T2
   id=lock3136940 mode=X associatedObjectId=72057594057392128
    owner-list
     owner id=process689978 mode=X
    waiter-list
     waiter id=process6891f8 mode=U requestType=wait
   keylock hobtid=72057594057457664 dbid=6 objectname=AdventureWorks2016.dbo.T1
   indexname=nci_T1_COL1 id=lock3136fc0 mode=X
   associatedObjectId=72057594057457664
    owner-list
     owner id=process6891f8 mode=X
    waiter-list
     waiter id=process689978 mode=U requestType=wait
```

###### Profiler Deadlock Graph Event

This is an event in SQL Profiler that presents a graphical depiction of the tasks and resources involved in a deadlock. The following example shows the output from SQL Profiler when the deadlock graph event is turned on.




For more information about the deadlock event, see Lock:Deadlock Event Class.

For more information about running the SQL Profiler deadlock graph, see Save Deadlock Graphs (SQL Server Profiler).

#### Handling Deadlocks

When an instance of the SQL Server Database Engine chooses a transaction as a deadlock victim, it terminates the current batch, rolls back the transaction, and returns error message 1205 to the application.

```
Your transaction (process ID #52) was deadlocked on {lock | communication buffer | thread} resources with
another process and has been chosen as the deadlock victim. Rerun your transaction.
```

Because any application submitting Transact-SQL queries can be chosen as the deadlock victim, applications should have an error handler that can trap error message 1205. If an application does not trap the error, the application can proceed unaware that its transaction has been rolled back and errors can occur.

Implementing an error handler that traps error message 1205 allows an application to handle the deadlock situation and take remedial action (for example, automatically resubmitting the query that was involved in the deadlock). By resubmitting the query automatically, the user does not need to know that a deadlock occurred.

The application should pause briefly before resubmitting its query. This gives the other transaction involved in the deadlock a chance to complete and release its locks that formed part of the deadlock cycle. This minimizes the likelihood of the deadlock reoccurring when the resubmitted query requests its locks.

#### Minimizing Deadlocks

Although deadlocks cannot be completely avoided, following certain coding conventions can minimize the chance of generating a deadlock. Minimizing deadlocks can increase transaction throughput and reduce system overhead because fewer transactions are:

- Rolled back, undoing all the work performed by the transaction.
- Resubmitted by applications because they were rolled back when deadlocked.

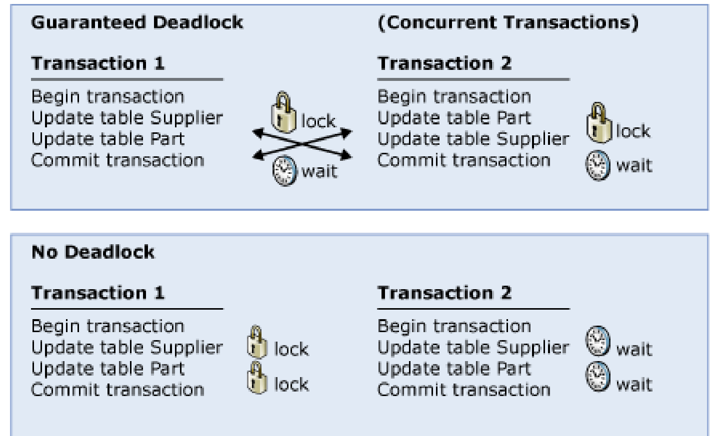
  To help minimize deadlocks:

- 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.

##### Access Objects in the same order

If all concurrent transactions access objects in the same order, deadlocks are less likely to occur. For example, if two concurrent transactions obtain a lock on the **Supplier** table and then on the **Part** table, one transaction is blocked on the **Supplier** table until the other transaction is completed. After the first transaction commits or rolls back, the second continues, and a deadlock does not occur. Using stored procedures for all data modifications can

standardize the order of accessing objects.




###### Avoid user interaction in Transactions

Avoid writing transactions that include user interaction, because the speed of batches running without user intervention is much faster than the speed at which a user must manually respond to queries, such as replying to a prompt for a parameter requested by an application. For example, if a transaction is waiting for user input and the user goes to lunch or even home for the weekend, the user delays the transaction from completing. This degrades system throughput because any locks held by the transaction are released only when the transaction is committed or rolled back. Even if a deadlock situation does not arise, other transactions accessing the same resources are blocked while waiting for the transaction to complete.

###### Keep Transactions short and in one batch

A deadlock typically occurs when several long-running transactions execute concurrently in the same database. The longer the transaction, the longer the exclusive or update locks are held, blocking other activity and leading to possible deadlock situations.

Keeping transactions in one batch minimizes network roundtrips during a transaction, reducing possible delays in completing the transaction and releasing locks.

###### 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.

> **NOTE**
>
> Some applications rely upon locking and blocking behavior of read committed isolation. For these applications, some change is required before this option can be enabled.

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. Only object locks can be partitioned.Object locks that have a subtype are not partitioned. For more information, see sys.dm_tran_locks (Transact-SQL).

#### 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.

#### Implementing and Monitoring Lock Partitioning

Lock partitioning is turned on by default for systems with 16 or more CPUs. When lock partitioning is enabled, an informational message is recorded in the SQL Server error log.

When acquiring locks on a partitioned resource:

- Only NL, SCH-S, IS, IU, and IX lock modes are acquired on a single partition.

- Shared (S), exclusive (X), and other locks in modes other than NL, SCH-S, IS, IU, and IX must be acquired on all partitions starting with partition ID 0 and following in partition ID order. These locks on a partitioned resource will use more memory than locks in the same mode on a non-partitioned resource since each partition is effectively a separate lock. The memory increase is determined by the number of partitions. The SQL Server lock counters in the Windows Performance Monitor will display information about memory used by partitioned and non-partitioned locks.

  A transaction is assigned to a partition when the transaction starts. For the transaction, all lock requests that can be partitioned use the partition assigned to that transaction. By this method, access to lock resources of the same object by different transactions is distributed across different partitions.

  The `resource_lock_partition` column in the `sys.dm_tran_locks` Dynamic Management View provides the lock partition ID for a lock partitioned resource. For more information, see sys.dm_tran_locks (Transact-SQL).

#### Working with Lock Partitioning

The following code examples illustrate lock partitioning. In the examples, two transactions are executed in two different sessions in order to show lock partitioning behavior on a computer system with 16 CPUs.

These Transact-SQL statements create test objects that are used in the examples that follow.

```
-- Create a test table.
CREATE TABLE TestTable
    (col1        int);
GO

-- Create a clustered index on the table.
CREATE CLUSTERED INDEX ci_TestTable
    ON TestTable (col1);
GO

-- Populate the table.
INSERT INTO TestTable VALUES (1);
GO
```

###### Example A

Session 1:

A `SELECT` statement is executed under a transaction. Because of the `HOLDLOCK` lock hint, this statement will acquire and retain an Intent shared (IS) lock on the table (for this illustration, row and page locks are ignored). The IS lock will be acquired only on the partition assigned to the transaction. For this example, it is assumed that the IS lock is acquired on partition ID 7.

```
-- Start a transaction.
BEGIN TRANSACTION
    -- This SELECT statement will acquire an IS lock on the table.
    SELECT col1
        FROM TestTable
        WITH (HOLDLOCK);
```

Session 2:

A transaction is started, and the `SELECT` statement running under this transaction will acquire and retain a shared (S) lock on the table. The S lock will be acquired on all partitions which results in multiple table locks, one for each partition. For example, on a 16-cpu system, 16 S locks will be issued across lock partition IDs 0-15. Because the S lock is compatible with the IS lock being held on partition ID 7 by the transaction in session 1, there is no blocking between transactions.

```
BEGIN TRANSACTION
    SELECT col1
        FROM TestTable
        WITH (TABLOCK, HOLDLOCK);
```

Session 1:

The following `SELECT` statement is executed under the transaction that is still active under session 1. Because of the exclusive (X) table lock hint, the transaction will attempt to acquire an X lock on the table. However, the S lock that is being held by the transaction in session 2 will block the X lock at partition ID 0.

```
SELECT col1
    FROM TestTable
    WITH (TABLOCKX);
```

###### Example B

Session 1:

A `SELECT` statement is executed under a transaction. Because of the `HOLDLOCK` lock hint, this statement will acquire and retain an Intent shared (IS) lock on the table (for this illustration, row and page locks are ignored). The IS lock

will be acquired only on the partition assigned to the transaction. For this example, it is assumed that the IS lock is acquired on partition ID 6.

```
-- Start a transaction.
BEGIN TRANSACTION
    -- This SELECT statement will acquire an IS lock on the table.
    SELECT col1
        FROM TestTable
        WITH (HOLDLOCK);
```

Session 2:

A `SELECT` statement is executed under a transaction. Because of the `TABLOCKX` lock hint, the transaction tries to acquire an exclusive (X) lock on the table. Remember that the X lock must be acquired on all partitions starting with partition ID 0. The X lock will be acquired on all partitions IDs 0-5 but will be blocked by the IS lock that is acquired on partition ID 6.

On partition IDs 7-15 that the X lock has not yet reached, other transactions can continue to acquire locks.

```
BEGIN TRANSACTION
    SELECT col1
        FROM TestTable
        WITH (TABLOCKX, HOLDLOCK);
```

# Row Versioning-based Isolation Levels in the SQL Server Database Engine

Starting with SQL Server 2005, the SQL Server Database Engine offers an implementation of an existing transaction isolation level, read committed, that provides a statement level snapshot using row versioning. SQL Server Database Engine also offers a transaction isolation level, snapshot, that provides a transaction level snapshot also using row versioning.

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 that require an earlier transactionally consistent state. 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`.

> **NOTE**
>
> For modification of large objects (LOBs), only the changed fragment is copied to the version store in `tempdb`.

Row versions are held long enough to satisfy the requirements of transactions running under row versioning-based isolation levels. The SQL Server Database Engine tracks the earliest useful transaction sequence number and periodically deletes all row versions stamped with transaction sequence numbers that are lower than the earliest useful sequence number.

When both database options are set to OFF, only rows modified by triggers or MARS sessions, or read by ONLINE index operations, are versioned. Those row versions are released when no longer needed. A background thread periodically executes to remove stale row versions.

> **NOTE**
>
> For short-running transactions, a version of a modified row may get cached in the buffer pool without getting written into the disk files of the `tempdb` database. If the need for the versioned row is short-lived, it will simply get dropped from the buffer pool and may not necessarily incur I/O overhead.

**Behavior when reading data**

When transactions running under row versioning-based isolation read data, the read operations do not acquire shared (S) locks on the data being read, and therefore do not block transactions that are modifying data. Also, the overhead of locking resources is minimized as the number of locks acquired is reduced. Read committed isolation using row versioning and snapshot isolation are designed to provide statement-level or transaction-level read consistencies of versioned data.

All queries, including transactions running under row versioning-based isolation levels, acquire Sch-S (schema stability) locks during compilation and execution. Because of this, queries are blocked when a concurrent transaction holds a Sch-M (schema modification) lock on the table. For example, a data definition language (DDL)

operation acquires a Sch-M lock before it modifies the schema information of the table. Query transactions, including those running under a row versioning-based isolation level, are blocked when attempting to acquire a Sch-S lock. Conversely, a query holding a Sch-S lock blocks a concurrent transaction that attempts to acquire a Sch-M lock.

When a transaction using the snapshot isolation level starts, the instance of the SQL Server Database Engine records all of the currently active transactions. When the snapshot transaction reads a row that has a version chain, the SQL Server Database Engine follows the chain and retrieves the row where the transaction sequence number is:

- Closest to but lower than the sequence number of the snapshot transaction reading the row.
- Not in the list of the transactions active when the snapshot transaction started.

  Read operations performed by a snapshot transaction retrieve the last version of each row that had been committed at the time the snapshot transaction started. This provides a transactionally consistent snapshot of the data as it existed at the start of the transaction.

  Read-committed transactions using row versioning operate in much the same way. The difference is that the read-committed transaction does not use its own transaction sequence number when choosing row versions. Each time a statement is started, the read-committed transaction reads the latest transaction sequence number issued for that instance of the SQL Server Database Engine. This is the transaction sequence number used to select the correct row versions for that statement. This allows read-committed transactions to see a snapshot of the data as it exists at the start of each statement.

> **NOTE**
>
> Even though read-committed transactions using row versioning provides a transactionally consistent view of the data at a statement level, row versions generated or accessed by this type of transaction are maintained until the transaction completes.

**Behavior when modifying data**

In a read-committed transaction using row versioning, the selection of rows to update is done using a blocking scan where an update (U) lock is taken on the data row as data values are read. This is the same as a read-committed transaction that does not use row versioning. If the data row does not meet the update criteria, the update lock is released on that row and the next row is locked and scanned.

Transactions running under snapshot isolation take an optimistic approach to data modification by acquiring locks on data before performing the modification only to enforce constraints. Otherwise, locks are not acquired on data until the data is to be modified. When a data row meets the update criteria, the snapshot transaction verifies that the data row has not been modified by a concurrent transaction that committed after the snapshot transaction began. If the data row has been modified outside of the snapshot transaction, an update conflict occurs and the snapshot transaction is terminated. The update conflict is handled by the SQL Server Database Engine and there is no way to disable the update conflict detection.

> **NOTE**
>
> Update operations running under snapshot isolation internally execute under read committed isolation when the snapshot transaction accesses any of the following:
>
> A table with a FOREIGN KEY constraint.
>
> A table that is referenced in the FOREIGN KEY constraint of another table.
>
> An indexed view referencing more than one table.
>
> However, even under these conditions the update operation will continue to verify that the data has not been modified by another transaction. If data has been modified by another transaction, the snapshot transaction encounters an update conflict and is terminated.

#### Behavior in summary

The following table summarizes the differences between snapshot isolation and read committed isolation using row versioning.

| PROPERTY | READ-COMMITTED ISOLATION LEVEL USING ROW VERSIONING | SNAPSHOT ISOLATION LEVEL |
| --- | --- | --- |
| The database option that must be set to ON to enable the required support. | READ_COMMITTED_SNAPSHOT | ALLOW_SNAPSHOT_ISOLATION |
| How a session requests the specific type of row versioning. | Use the default read-committed isolation level, or run the SET TRANSACTION ISOLATION LEVEL statement to specify the READ COMMITTED isolation level. This can be done after the transaction starts. | Requires the execution of SET TRANSACTION ISOLATION LEVEL to specify the SNAPSHOT isolation level before the start of the transaction. |
| The version of data read by statements. | All data that was committed before the start of each statement. | All data that was committed before the start of each transaction. |
| How updates are handled. | Reverts from row versions to actual data to select rows to update and uses update locks on the data rows selected. Acquires exclusive locks on actual data rows to be modified. No update conflict detection. | Uses row versions to select rows to update. Tries to acquire an exclusive lock on the actual data row to be modified, and if the data has been modified by another transaction, an update conflict occurs and the snapshot transaction is terminated. |
| Update conflict detection. | None. | Integrated support. Cannot be disabled. |

#### Row Versioning resource usage

The row versioning framework supports the following features available in SQL Server:

- Triggers
- Multiple Active Results Sets (MARS)
- Online indexing

  The row versioning framework also supports the following row versioning-based transaction isolation levels, which by default are not enabled:

- When the `READ_COMMITTED_SNAPSHOT` database option is ON, `READ_COMMITTED` transactions provide statement-level read consistency using row versioning.

- When the `ALLOW_SNAPSHOT_ISOLATION` database option is ON, `SNAPSHOT` transactions provide transaction-level read consistency using row versioning.

  Row versioning-based isolation levels reduce the number of locks acquired by transaction by eliminating the use of shared locks on read operations. This increases system performance by reducing the resources used to manage locks. Performance is also increased by reducing the number of times a transaction is blocked by locks acquired by other transactions.

  Row versioning-based isolation levels increase the resources needed by data modifications. Enabling these options causes all data modifications for the database to be versioned. A copy of the data before modification is stored in tempdb even when there are no active transactions using row versioning-based isolation. The data after modification includes a pointer to the versioned data stored in tempdb. For large objects, only part of the object that changed is copied to tempdb.

**Space used in TempDB**

For each instance of the SQL Server Database Engine, tempdb must have enough space to hold the row versions generated for every database in the instance. The database administrator must ensure that TempDB has ample space to support the version store. There are two version stores in TempDB:

- The online index build version store is used for online index builds in all databases.
- The common version store is used for all other data modification operations in all databases.

  Row versions must be stored for as long as an active transaction needs to access it. Once every minute, a background thread removes row versions that are no longer needed and frees up the version space in TempDB. A long-running transaction prevents space in the version store from being released if it meets any of the following conditions:

- It uses row versioning-based isolation.
- It uses triggers, MARS, or online index build operations.
- It generates row versions.

> **NOTE**
>
> When a trigger is invoked inside a transaction, the row versions created by the trigger are maintained until the end of the transaction, even though the row versions are no longer needed after the trigger completes. This also applies to read-committed transactions that use row versioning. With this type of transaction, a transactionally consistent view of the database is needed only for each statement in the transaction. This means that the row versions created for a statement in the transaction are no longer needed after the statement completes. However, row versions created by each statement in the transaction are maintained until the transaction completes.

When TempDB runs out of space, the SQL Server Database Engine forces the version stores to shrink. During the shrink process, the longest running transactions that have not yet generated row versions are marked as victims. A message 3967 is generated in the error log for each victim transaction. If a transaction is marked as a victim, it can no longer read the row versions in the version store. When it attempts to read row versions, message 3966 is generated and the transaction is rolled back. If the shrinking process succeeds, space becomes available in tempdb. Otherwise, tempdb runs out of space and the following occurs:

- Write operations continue to execute but do not generate versions. An information message (3959) appears in the error log, but the transaction that writes data is not affected.
- Transactions that attempt to access row versions that were not generated because of a tempdb full rollback terminate with an error 3958.

**Space used in data rows**

Each database row may use up to 14 bytes at the end of the row for row versioning information. The row

versioning information contains the transaction sequence number of the transaction that committed the version and the pointer to the versioned row. These 14 bytes are added the first time the row is modified, or when a new row is inserted, under any of these conditions:

- `READ_COMMITTED_SNAPSHOT` or `ALLOW_SNAPSHOT_ISOLATION` options are ON.
- The table has a trigger.
- Multiple Active Results Sets (MARS) is being used.
- Online index build operations are currently running on the table.

  These 14 bytes are removed from the database row the first time the row is modified under all of these conditions:

- `READ_COMMITTED_SNAPSHOT` and `ALLOW_SNAPSHOT_ISOLATION` options are OFF.

- The trigger no longer exists on the table.
- MARS is not being used.
- Online index build operations are not currently running.

  If you use any of the row versioning features, you might need to allocate additional disk space for the database to accommodate the 14 bytes per database row. Adding the row versioning information can cause index page splits or the allocation of a new data page if there is not enough space available on the current page. For example, if the average row length is 100 bytes, the additional 14 bytes cause an existing table to grow up to 14 percent.

  Decreasing the fill factor might help to prevent or decrease fragmentation of index pages. To view fragmentation information for the data and indexes of a table or view, you can use sys.dm_db_index_physical_stats.

**Space used in Large Objects**

The SQL Server Database Engine supports six data types that can hold large strings up to 2 gigabytes (GB) in length: `nvarchar(max)`, `varchar(max)`, `varbinary(max)`, `ntext`, `text`, and `image`. Large strings stored using these data types are stored in a series of data fragments that are linked to the data row. Row versioning information is stored in each fragment used to store these large strings. Data fragments are a collection of pages dedicated to large objects in a table.

As new large values are added to a database, they are allocated using a maximum of 8040 bytes of data per fragment. Earlier versions of the SQL Server Database Engine stored up to 8080 bytes of `ntext`, `text`, or `image` data per fragment.

Existing `ntext`, `text`, and `image` large object (LOB) data is not updated to make space for the row versioning information when a database is upgraded to SQL Server from an earlier version of SQL Server. However, the first time the LOB data is modified, it is dynamically upgraded to enable storage of versioning information. This will happen even if row versions are not generated. After the LOB data is upgraded, the maximum number of bytes stored per fragment is reduced from 8080 bytes to 8040 bytes. The upgrade process is equivalent to deleting the LOB value and reinserting the same value. The LOB data is upgraded even if only one byte is modified. This is a one-time operation for each `ntext`, `text`, or `image` column, but each operation may generate a large amount of page allocations and I/O activity depending upon the size of the LOB data. It may also generate a large amount of logging activity if the modification is fully logged. WRITETEXT and UPDATETEXT operations are minimally logged if database recovery mode is not set to FULL.

The `nvarchar(max)`, `varchar(max)`, and `varbinary(max)` data types are not available in earlier versions of SQL Server. Therefore, they have no upgrade issues.

Enough disk space should be allocated to accommodate this requirement.

**Monitoring Row Versioning and the Version Store**

For monitoring row versioning, version store, and snapshot isolation processes for performance and problems, SQL Server provides tools in the form of Dynamic Management Views (DMVs) and performance counters in Windows System Monitor.

###### DMVs

The following DMVs provide information about the current system state of tempdb and the version store, as well as transactions using row versioning.

sys.dm_db_file_space_usage. Returns space usage information for each file in the database. For more information, see sys.dm_db_file_space_usage (Transact-SQL).

sys.dm_db_session_space_usage. Returns page allocation and deallocation activity by session for the database. For more information, see sys.dm_db_session_space_usage (Transact-SQL).

sys.dm_db_task_space_usage. Returns page allocation and deallocation activity by task for the database. For more information, see sys.dm_db_task_space_usage (Transact-SQL).

sys.dm_tran_top_version_generators. Returns a virtual table for the objects producing the most versions in the version store. It groups the top 256 aggregated record lengths by database_id and rowset_id. Use this function to find the largest consumers of the version store. For more information, see sys.dm_tran_top_version_generators (Transact-SQL).

sys.dm_tran_version_store. Returns a virtual table that displays all version records in the common version store. For more information, see sys.dm_tran_version_store (Transact-SQL).

sys.dm_tran_version_store_space_usage. Returns a virtual table that displays the total space in tempdb used by version store records for each database. For more information, see sys.dm_tran_version_store_space_usage (Transact-SQL).

> **NOTE**
>
> sys.dm_tran_top_version_generators and sys.dm_tran_version_store are potentially very expensive functions to run, since both query the entire version store, which could be very large.
> sys.dm_tran_version_store_space_usage is efficient and not expensive to run, as it does not navigate through individual version store records and returns aggregated version store space consumed in tempdb per database

sys.dm_tran_active_snapshot_database_transactions. Returns a virtual table for all active transactions in all databases within the SQL Server instance that use row versioning. System transactions do not appear in this DMV. For more information, see sys.dm_tran_active_snapshot_database_transactions (Transact-SQL).

sys.dm_tran_transactions_snapshot. Returns a virtual table that displays snapshots taken by each transaction. The snapshot contains the sequence number of the active transactions that use row versioning. For more information, see sys.dm_tran_transactions_snapshot (Transact-SQL).

sys.dm_tran_current_transaction. Returns a single row that displays row versioning-related state information of the transaction in the current session. For more information, see sys.dm_tran_current_transaction (Transact-SQL).

sys.dm_tran_current_snapshot. Returns a virtual table that displays all active transactions at the time the current snapshot isolation transaction starts. If the current transaction is using snapshot isolation, this function returns no rows. sys.dm_tran_current_snapshot is similar to sys.dm_tran_transactions_snapshot, except that it returns only the active transactions for the current snapshot. For more information, see sys.dm_tran_current_snapshot (Transact-SQL).

###### Performance Counters

SQL Server performance counters provide information about the system performance impacted by SQL Server processes. The following performance counters monitor tempdb and the version store, as well as transactions using row versioning. The performance counters are contained in the SQLServer:Transactions performance object.

**Free Space in tempdb (KB)**. Monitors the amount, in kilobytes (KB), of free space in the tempdb database. There must be enough free space in tempdb to handle the version store that supports snapshot isolation.

The following formula provides a rough estimate of the size of the version store. For long-running transactions, it may be useful to monitor the generation and cleanup rate to estimate the maximum size of the version store.

[size of common version store] = 2 * [version store data generated per minute] * [longest running time (minutes) of the transaction]

The longest running time of transactions should not include online index builds. Because these operations may take a long time on very large tables, online index builds use a separate version store. The approximate size of the online index build version store equals the amount of data modified in the table, including all indexes, while the online index build is active.

**Version Store Size (KB)**. Monitors the size in KB of all version stores. This information helps determine the amount of space needed in the tempdb database for the version store. Monitoring this counter over a period of time provides a useful estimate of additional space needed for tempdb.

`Version Generation rate (KB/s)`. Monitors the version generation rate in KB per second in all version stores.

`Version Cleanup rate (KB/s)`. Monitors the version cleanup rate in KB per second in all version stores.

> **NOTE**
>
> Information from Version Generation rate (KB/s) and Version Cleanup rate (KB/s) can be used to predict tempdb space requirements.

**Version Store unit count**. Monitors the count of version store units.

**Version Store unit creation**. Monitors the total number of version store units created to store row versions since the instance was started.

**Version Store unit truncation**. Monitors the total number of version store units truncated since the instance was started. A version store unit is truncated when SQL Server determines that none of the version rows stored in the version store unit are needed to run active transactions.

**Update conflict ratio**. Monitors the ratio of update snapshot transaction that have update conflicts to the total number of update snapshot transactions.

**Longest Transaction Running Time**. Monitors the longest running time in seconds of any transaction using row versioning. This can be used to determine if any transaction is running for an unreasonable amount of time.

**Transactions**. Monitors the total number of active transactions. This does not include system transactions.

`Snapshot Transactions`. Monitors the total number of active snapshot transactions.

`Update Snapshot Transactions`. Monitors the total number of active snapshot transactions that perform update operations.

`NonSnapshot Version Transactions`. Monitors the total number of active non-snapshot transactions that generate version records.

> **NOTE**
>
> The sum of Update Snapshot Transactions and NonSnapshot Version Transactions represents the total number of transactions that participate in version generation. The difference of Snapshot Transactions and Update Snapshot Transactions reports the number of read-only snapshot transactions.

### Row Versioning-based Isolation Level Example

The following examples show the differences in behavior between snapshot isolation transactions and read-committed transactions that use row versioning.

#### A. Working with snapshot isolation

In this example, a transaction running under snapshot isolation reads data that is then modified by another transaction. The snapshot transaction does not block the update operation executed by the other transaction, and it continues to read data from the versioned row, ignoring the data modification. However, when the snapshot transaction attempts to modify the data that has already been modified by the other transaction, the snapshot transaction generates an error and is terminated.

On session 1:

```
USE AdventureWorks2016;
GO

-- Enable snapshot isolation on the database.
ALTER DATABASE AdventureWorks2016
    SET ALLOW_SNAPSHOT_ISOLATION ON;
GO

-- Start a snapshot transaction
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
GO

BEGIN TRANSACTION;
    -- This SELECT statement will return
    -- 48 vacation hours for the employee.
    SELECT BusinessEntityID, VacationHours
        FROM HumanResources.Employee
        WHERE BusinessEntityID = 4;
```

On session 2:

```
USE AdventureWorks2016;
GO

-- Start a transaction.
BEGIN TRANSACTION;
    -- Subtract a vacation day from employee 4.
    -- Update is not blocked by session 1 since
    -- under snapshot isolation shared locks are
    -- not requested.
    UPDATE HumanResources.Employee
        SET VacationHours = VacationHours - 8
        WHERE BusinessEntityID = 4;

    -- Verify that the employee now has 40 vacation hours.
    SELECT VacationHours
        FROM HumanResources.Employee
        WHERE BusinessEntityID = 4;
```

On session 1:

```
    -- Reissue the SELECT statement - this shows
    -- the employee having 48 vacation hours.  The
    -- snapshot transaction is still reading data from
    -- the versioned row.
    SELECT BusinessEntityID, VacationHours
        FROM HumanResources.Employee
        WHERE BusinessEntityID = 4;
```

On session 2:

```
-- Commit the transaction; this commits the data
-- modification.
COMMIT TRANSACTION;
GO
```

On session 1:

```
    -- Reissue the SELECT statement - this still
    -- shows the employee having 48 vacation hours
    -- even after the other transaction has committed
    -- the data modification.
    SELECT BusinessEntityID, VacationHours
        FROM HumanResources.Employee
        WHERE BusinessEntityID = 4;

    -- Because the data has been modified outside of the
    -- snapshot transaction, any further data changes to
    -- that data by the snapshot transaction will cause
    -- the snapshot transaction to fail. This statement
    -- will generate a 3960 error and the transaction will
    -- terminate.
    UPDATE HumanResources.Employee
        SET SickLeaveHours = SickLeaveHours - 8
        WHERE BusinessEntityID = 4;

-- Undo the changes to the database from session 1.
-- This will not undo the change from session 2.
ROLLBACK TRANSACTION
GO
```

#### B. Working with read-committed using row versioning

In this example, a read-committed transaction using row versioning runs concurrently with another transaction. The read-committed transaction behaves differently than a snapshot transaction. Like a snapshot transaction, the read-committed transaction will read versioned rows even after the other transaction has modified data. However, unlike a snapshot transaction, the read-committed transaction will:

- Read the modified data after the other transaction commits the data changes.
- Be able to update the data modified by the other transaction where the snapshot transaction could not.

  On session 1:

```
USE AdventureWorks2016;  -- Or any earlier version of the AdventureWorks database.
GO

-- Enable READ_COMMITTED_SNAPSHOT on the database.
-- For this statement to succeed, this session
-- must be the only connection to the AdventureWorks2016
-- database.
ALTER DATABASE AdventureWorks2016
    SET READ_COMMITTED_SNAPSHOT ON;
GO

-- Start a read-committed transaction
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
GO

BEGIN TRANSACTION;
    -- This SELECT statement will return
    -- 48 vacation hours for the employee.
    SELECT BusinessEntityID, VacationHours
        FROM HumanResources.Employee
        WHERE BusinessEntityID = 4;
```

On session 2:

```
USE AdventureWorks2016;
GO

-- Start a transaction.
BEGIN TRANSACTION;
    -- Subtract a vacation day from employee 4.
    -- Update is not blocked by session 1 since
    -- under read-committed using row versioning shared locks are
    -- not requested.
    UPDATE HumanResources.Employee
        SET VacationHours = VacationHours - 8
        WHERE BusinessEntityID = 4;

    -- Verify that the employee now has 40 vacation hours.
    SELECT VacationHours
        FROM HumanResources.Employee
        WHERE BusinessEntityID = 4;
```

On session 1:

```
    -- Reissue the SELECT statement - this still shows
    -- the employee having 48 vacation hours.  The
    -- read-committed transaction is still reading data
    -- from the versioned row and the other transaction
    -- has not committed the data changes yet.
    SELECT BusinessEntityID, VacationHours
        FROM HumanResources.Employee
        WHERE BusinessEntityID = 4;
```

On session 2:

```
-- Commit the transaction.
COMMIT TRANSACTION;
GO
```

On session 1:

```
    -- Reissue the SELECT statement which now shows the
    -- employee having 40 vacation hours.  Being
    -- read-committed, this transaction is reading the
    -- committed data. This is different from snapshot
    -- isolation which reads from the versioned row.
    SELECT BusinessEntityID, VacationHours
        FROM HumanResources.Employee
        WHERE BusinessEntityID = 4;

    -- This statement, which caused the snapshot transaction
    -- to fail, will succeed with read-committed using row versioning.
    UPDATE HumanResources.Employee
        SET SickLeaveHours = SickLeaveHours - 8
        WHERE BusinessEntityID = 4;

-- Undo the changes to the database from session 1.
-- This will not undo the change from session 2.
ROLLBACK TRANSACTION;
GO
```

**Enabling Row Versioning-Based Isolation Levels**

Database administrators control the database-level settings for row versioning by using the `READ_COMMITTED_SNAPSHOT` and `ALLOW_SNAPSHOT_ISOLATION` database options in the ALTER DATABASE statement.

When the `READ_COMMITTED_SNAPSHOT` database option is set ON, the mechanisms used to support the option are activated immediately. When setting the READ_COMMITTED_SNAPSHOT option, only the connection executing the `ALTER DATABASE` command is allowed in the database. There must be no other open connection in the database until ALTER DATABASE is complete. The database does not have to be in single-user mode.

The following Transact-SQL statement enables `READ_COMMITTED_SNAPSHOT`:

```
ALTER DATABASE AdventureWorks2016
    SET READ_COMMITTED_SNAPSHOT ON;
```

When the `ALLOW_SNAPSHOT_ISOLATION` database option is set ON, the instance of the SQL Server Database Engine does not generate row versions for modified data until all active transactions that have modified data in the database complete. If there are active modification transactions, SQL Server sets the state of the option to `PENDING_ON`. After all of the modification transactions complete, the state of the option is changed to ON. Users cannot start a snapshot transaction in that database until the option is fully ON. The database passes through a PENDING_OFF state when the database administrator sets the `ALLOW_SNAPSHOT_ISOLATION` option to OFF.

The following Transact-SQL statement will enable ALLOW_SNAPSHOT_ISOLATION:

```
ALTER DATABASE AdventureWorks2016
    SET ALLOW_SNAPSHOT_ISOLATION ON;
```

The following table lists and describes the states of the ALLOW_SNAPSHOT_ISOLATION option. Using ALTER DATABASE with the ALLOW_SNAPSHOT_ISOLATION option does not block users who are currently accessing the database data.

| STATE OF SNAPSHOT ISOLATION FRAMEWORK FOR CURRENT DATABASE | DESCRIPTION |
| --- | --- |
| OFF | The support for snapshot isolation transactions is not activated. No snapshot isolation transactions are allowed. |

| STATE OF SNAPSHOT ISOLATION FRAMEWORK FOR CURRENT DATABASE | DESCRIPTION |
|---|---|
| PENDING_ON | The support for snapshot isolation transactions is in transition state (from OFF to ON). Open transactions must complete.<br><br>No snapshot isolation transactions are allowed. |
| ON | The support for snapshot isolation transactions is activated.<br><br>Snapshot transactions are allowed. |
| PENDING_OFF | The support for snapshot isolation transactions is in transition state (from ON to OFF).<br><br>Snapshot transactions started after this time cannot access this database. Update transactions still pay the cost of versioning in this database. Existing snapshot transactions can still access this database without a problem. The state PENDING_OFF does not become OFF until all snapshot transactions that were active when the database snapshot isolation state was ON finish. |

Use the `sys.databases` catalog view to determine the state of both row versioning database options.

All updates to user tables and some system tables stored in master and msdb generate row versions.

The `ALLOW_SNAPSHOT_ISOLATION` option is automatically set ON in the master and msdb databases, and cannot be disabled.

Users cannot set the `READ_COMMITTED_SNAPSHOT` option ON in master, tempdb, or msdb.

**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. Any application accessing objects from enabled databases can run queries using the following isolation levels:

- Read-committed that uses row versioning by setting the `READ_COMMITTED_SNAPSHOT` database option to `ON` as shown in the following code example:

  ```
  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` as shown in the following code example:

  ```
  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. For

example, the following code example shows a `SELECT` statement that joins two tables while running under a snapshot transaction. One table belongs to a database in which snapshot isolation is not enabled. When the `SELECT` statement runs under snapshot isolation, it fails to execute successfully.

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRAN
    SELECT t1.col5, t2.col5
        FROM Table1 as t1
        INNER JOIN SecondDB.dbo.Table2 as t2
            ON t1.col1 = t2.col2;
```

The following code example shows the same `SELECT` statement that has been modified to change the transaction isolation level to read-committed. Because of this change, the `SELECT` statement executes successfully.

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRAN
    SELECT t1.col5, t2.col5
        FROM Table1 as t1
        WITH (READCOMMITTED)
        INNER JOIN SecondDB.dbo.Table2 as t2
            ON t1.col1 = t2.col2;
```

#### 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.

  For example, a database administrator executes the following `ALTER INDEX` statement.

```
USE AdventureWorks2016;
GO
ALTER INDEX AK_Employee_LoginID
    ON HumanResources.Employee REBUILD;
GO
```

Any snapshot transaction that is active when the `ALTER INDEX` statement is executed receives an error if it attempts to reference the `HumanResources.Employee` table after the `ALTER INDEX` statement is executed. Read-committed transactions using row versioning are not affected.

> **NOTE**
>
> BULK INSERT operations may cause changes to target table metadata (for example, when disabling constraint checks). When this happens, concurrent snapshot isolation transactions accessing bulk inserted tables fail.

# Customizing Locking and Row Versioning

**Customizing the Lock Time-Out**

When an instance of the Microsoft SQL Server Database Engine cannot grant a lock to a transaction because another transaction already owns a conflicting lock on the resource, the first transaction becomes blocked waiting for the existing lock to be released. By default, there is no mandatory time-out period and no way to test whether a resource is locked before locking it, except to attempt to access the data (and potentially get blocked indefinitely).

> **NOTE**
>
> In SQL Server, use the **sys.dm_os_waiting_tasks** dynamic management view to determine whether a process is being blocked and who is blocking it. In earlier versions of SQL Server, use the **sp_who** system stored procedure.

The `LOCK_TIMEOUT` setting allows an application to set a 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 canceled automatically, and error message 1222 (`Lock request time-out period exceeded`) is returned to the application. Any transaction containing the statement, however, is not rolled back or canceled by SQL Server. Therefore, the application must have an error handler that can trap error message 1222. If an application does not trap the error, the application can proceed unaware that an individual statement within a transaction has been canceled, and errors can occur because statements later in the transaction might depend on the statement that was never executed.

Implementing an error handler that traps error message 1222 allows an application to handle the time-out situation and take remedial action, such as: automatically resubmitting the statement that was blocked or rolling back the entire transaction.

To determine the current `LOCK_TIMEOUT` setting, execute the `@@LOCK_TIMEOUT` function:

```
SELECT @@lock_timeout;
GO
```

**Customizing Transaction Isolation Level**

READ COMMITTED is the default isolation level for the Microsoft SQL Server Database Engine. If an application must operate at a different isolation level, it can use the following methods to set the isolation level:

- Run the SET TRANSACTION ISOLATION LEVEL statement.
- ADO.NET applications that use the System.Data.SqlClient managed namespace can specify an *IsolationLevel*

option by using the SqlConnection.BeginTransaction method.

- Applications that use ADO can set the `Autocommit Isolation Levels` property.
- When starting a transaction, applications using OLE DB can call ITransactionLocal::StartTransaction with *isoLevel* set to the desired transaction isolation level. When specifying the isolation level in autocommit mode, applications that use OLE DB can set the DBPROPSET_SESSION property DBPROP_SESS_AUTOCOMMITISOLEVELS to the desired transaction isolation level.
- Applications that use ODBC can set the SQL_COPT_SS_TXN_ISOLATION attribute by using SQLSetConnectAttr.

When the isolation level is specified, the locking behavior for all queries and data manipulation language (DML) statements in the SQL Server session operates at that isolation level. The isolation level remains in effect until the session terminates or until the isolation level is set to another level.

The following example sets the `SERIALIZABLE` isolation level:

```
USE AdventureWorks2016;
GO
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
GO
BEGIN TRANSACTION;
SELECT BusinessEntityID
    FROM HumanResources.Employee;
GO
```

The isolation level can be overridden for individual query or DML statements, if necessary, by specifying a table-level hint. Specifying a table-level hint does not affect other statements in the session. We recommend that table-level hints be used to change the default behavior only when absolutely necessary.

The SQL Server Database Engine might have to acquire locks when reading metadata even when the isolation level is set to a level where share locks are not requested when reading data. For example, a transaction running at the read-uncommitted isolation level does not acquire share locks when reading data, but might sometime request locks when reading a system catalog view. This means it is possible for a read uncommitted transaction to cause blocking when querying a table when a concurrent transaction is modifying the metadata of that table.

To determine the transaction isolation level currently set, use the `DBCC USEROPTIONS` statement as shown in the following example. The result set may vary from the result set on your system.

```
USE AdventureWorks2016;
GO
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
GO
DBCC USEROPTIONS;
GO
```

Here is the result set.

```
Set Option                  Value
--------------------------- -------------------------------------------
textsize                    2147483647
language                    us_english
dateformat                  mdy
datefirst                   7
...                         ...
Isolation level             repeatable read

(14 row(s) affected)

DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

### Locking Hints

Locking hints can be specified for individual table references in the SELECT, INSERT, UPDATE, and DELETE statements. The hints specify the type of locking or row versioning the instance of the SQL Server Database Engine uses for the table data. Table-level locking hints can be used when a finer control of the types of locks acquired on an object is required. These locking hints override the current transaction isolation level for the session.

For more information about the specific locking hints and their behaviors, see Table Hints (Transact-SQL).

> **NOTE**
>
> The SQL Server Database Engine query optimizer almost always chooses the correct locking level. We recommend that table-level locking hints be used to change the default locking behavior only when necessary. Disallowing a locking level can adversely affect concurrency.

The SQL Server Database Engine might have to acquire locks when reading metadata, even when processing a select with a locking hint that prevents requests for share locks when reading data. For example, a `SELECT` using the `NOLOCK` hint does not acquire share locks when reading data, but might sometime request locks when reading a system catalog view. This means it is possible for a `SELECT` statement using `NOLOCK` to be blocked.

As shown in the following example, if the transaction isolation level is set to `SERIALIZABLE`, and the table-level locking hint `NOLOCK` is used with the `SELECT` statement, key-range locks typically used to maintain serializable transactions are not taken.

```
USE AdventureWorks2016;
GO
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
GO
BEGIN TRANSACTION;
GO
SELECT JobTitle
    FROM HumanResources.Employee WITH (NOLOCK);
GO

-- Get information about the locks held by
-- the transaction.
SELECT
        resource_type,
        resource_subtype,
        request_mode
    FROM sys.dm_tran_locks
    WHERE request_session_id = @@spid;

-- End the transaction.
ROLLBACK;
GO
```

The only lock taken that references *HumanResources.Employee* is a schema stability (Sch-S) lock. In this case, serializability is no longer guaranteed.

In SQL Server 2017, the `LOCK_ESCALATION` option of `ALTER TABLE` can disfavor table locks, and enable HoBT locks on partitioned tables. This option is not a locking hint, but can but used to reduce lock escalation. For more information, see ALTER TABLE (Transact-SQL).

**Customizing Locking for an Index**

The SQL Server Database Engine uses a dynamic locking strategy that automatically chooses the best locking granularity for queries in most cases. We recommend that you do not override the default locking levels, which have page and row locking on, unless table or index access patterns are well understood and consistent, and there is a resource contention problem to solve. Overriding a locking level can significantly impede concurrent access to a table or index. For example, specifying only table-level locks on a large table that users access heavily can cause bottlenecks because users must wait for the table-level lock to be released before accessing the table.

There are a few cases where disallowing page or row locking can be beneficial, if the access patterns are well understood and consistent. For example, a database application uses a lookup table that is updated weekly in a batch process. Concurrent readers access the table with a shared (S) lock and the weekly batch update accesses the table with an exclusive (X) lock. Turning off page and row locking on the table reduces the locking overhead throughout the week by allowing readers to concurrently access the table through shared table locks. When the batch job runs, it can complete the update efficiently because it obtains an exclusive table lock.

Turning off page and row locking might or might not be acceptable because the weekly batch update will block the concurrent readers from accessing the table while the update runs. If the batch job only changes a few rows or pages, you can change the locking level to allow row or page level locking, which will enable other sessions to read from the table without blocking. If the batch job has a large number of updates, obtaining an exclusive lock on the table may be the best way to ensure the batch job finishes efficiently.

Occasionally a deadlock occurs when two concurrent operations acquire row locks on the same table and then block because they both need to lock the page. Disallowing row locks forces one of the operations to wait, avoiding the deadlock.

The granularity of locking used on an index can be set using the `CREATE INDEX` and `ALTER INDEX` statements. The lock settings apply to both the index pages and the table pages. In addition, the `CREATE TABLE` and `ALTER TABLE` statements can be used to set locking granularity on `PRIMARY KEY` and `UNIQUE` constraints. For backwards

compatibility, the `sp_indexoption` system stored procedure can also set the granularity. To display the current locking option for a given index, use the `INDEXPROPERTY` function. Page-level locks, row-level locks, or a combination of page-level and row-level locks can be disallowed for a given index.

| DISALLOWED LOCKS | INDEX ACCESSED BY |
| --- | --- |
| Page level | Row-level and table-level locks |
| Row level | Page-level and table-level locks |
| Page level and row level | Table-level locks |

# 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 following example shows the intended use of nested transactions. The procedure *TransProc* enforces its transaction regardless of the transaction mode of any process that executes it. If *TransProc* is called when a transaction is active, the nested transaction in *TransProc* is largely ignored, and its `INSERT` statements are committed or rolled back based on the final action taken for the outer transaction. If `TransProc` is executed by a process that does not have an outstanding transaction, the `COMMIT TRANSACTION` at the end of the procedure effectively commits the `INSERT` statements.

```
SET QUOTED_IDENTIFIER OFF;
GO
SET NOCOUNT OFF;
GO
CREATE TABLE TestTrans(Cola INT PRIMARY KEY,
               Colb CHAR(3) NOT NULL);
GO
CREATE PROCEDURE TransProc @PriKey INT, @CharCol CHAR(3) AS
BEGIN TRANSACTION InProc
INSERT INTO TestTrans VALUES (@PriKey, @CharCol)
INSERT INTO TestTrans VALUES (@PriKey + 1, @CharCol)
COMMIT TRANSACTION InProc;
GO
/* Start a transaction and execute TransProc. */
BEGIN TRANSACTION OutOfProc;
GO
EXEC TransProc 1, 'aaa';
GO
/* Roll back the outer transaction, this will
   roll back TransProc's nested transaction. */
ROLLBACK TRANSACTION OutOfProc;
GO
EXECUTE TransProc 3,'bbb';
GO
/* The following SELECT statement shows only rows 3 and 4 are
   still in the table. This indicates that the commit
   of the inner transaction from the first EXECUTE statement of
   TransProc was overridden by the subsequent rollback. */
SELECT * FROM TestTrans;
GO
```

Committing inner transactions is ignored by the SQL Server Database Engine. 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.

Each call to `COMMIT TRANSACTION` or `COMMIT WORK` applies to the last executed `BEGIN TRANSACTION`. If the `BEGIN TRANSACTION` statements are nested, then a `COMMIT` statement applies only to the last nested transaction, which is the innermost transaction. Even if a `COMMIT TRANSACTION` *transaction_name* statement within a nested transaction refers to the transaction name of the outer transaction, the commit applies only to the innermost transaction.

It is not legal for the *transaction_name* parameter of a `ROLLBACK TRANSACTION` statement to refer to the inner transactions of a set of named nested transactions. *transaction_name* can refer only to the transaction name of the outermost transaction. If a ROLLBACK TRANSACTION *transaction_name* statement using the name of the outer transaction is executed at any level of a set of nested transactions, all of the nested transactions are rolled back. If a `ROLLBACK WORK` or `ROLLBACK TRANSACTION` statement without a *transaction_name* parameter is executed at any level of a set of nested transaction, it rolls back all of the nested transactions, including the outermost transaction.

The `@@TRANCOUNT` function records the current transaction nesting level. Each `BEGIN TRANSACTION` statement increments `@@TRANCOUNT` by one. Each `COMMIT TRANSACTION` or `COMMIT WORK` statement decrements `@@TRANCOUNT` by one. A `ROLLBACK WORK` or a `ROLLBACK TRANSACTION` statement that does not have a transaction name rolls back all nested transactions and decrements `@@TRANCOUNT` to 0. A `ROLLBACK TRANSACTION` that uses the transaction name of the outermost transaction in a set of nested transactions rolls back all of the nested transactions and decrements `@@TRANCOUNT` to 0. When you are unsure if you are already in a transaction, `SELECT @@TRANCOUNT` to determine if it is 1 or more. If `@@TRANCOUNT` is 0, you are not in a transaction.

#### Using Bound Sessions

Bound sessions ease the coordination of actions across multiple sessions on the same server. Bound sessions allow two or more sessions to share the same transaction and locks, and can work on the same data without lock conflicts. Bound sessions can be created from multiple sessions within the same application or from multiple applications with separate sessions.

To participate in a bound session, a session calls `sp_getbindtoken` or `srv_getbindtoken` (through Open Data Services) to get a bind token. A bind token is a character string that uniquely identifies each bound transaction. The bind token is then sent to the other sessions to be bound with the current session. The other sessions bind to the transaction by calling **sp_bindsession**, using the bind token received from the first session.

> **NOTE**
>
> A session must have an active user transaction in order for `sp_getbindtoken` or `srv_getbindtoken` to succeed.

Bind tokens must be transmitted from the application code that makes the first session to the application code that subsequently binds their sessions to the first session. There is no Transact-SQL statement or API function that an application can use to get the bind token for a transaction started by another process. Some of the methods that can be used to transmit a bind token include the following:

- If the sessions are all initiated from the same application process, bind tokens can be stored in global memory or passed into functions as a parameter.
- If the sessions are made from separate application processes, bind tokens can be transmitted using interprocess communication (IPC), such as a remote procedure call (RPC) or dynamic data exchange (DDE).
- Bind tokens can be stored in a table in an instance of the SQL Server Database Engine that can be read by processes wanting to bind to the first session.

  Only one session in a set of bound sessions can be active at any time. If one session is executing a statement on the instance or has results pending from the instance, no other session bound to it can access

the instance until the current session finishes processing or cancels the current statement. If the instance is busy processing a statement from another of the bound sessions, an error occurs indicating that the transaction space is in use and the session should retry later.

When you bind sessions, each session retains its isolation level setting. Using SET TRANSACTION ISOLATION LEVEL to change the isolation level setting of one session does not affect the setting of any other session bound to it.

##### Types of Bound Sessions

The two types of bound sessions are local and distributed.

- **Local bound session**
  Allows bound sessions to share the transaction space of a single transaction in a single instance of the SQL Server Database Engine.

- **Distributed bound session**
  Allows bound sessions to share the same transaction across two or more instances until the entire transaction is either committed or rolled back by using Microsoft Distributed Transaction Coordinator (MS DTC).

  Distributed bound sessions are not identified by a character string bind token; they are identified by distributed transaction identification numbers. If a bound session is involved in a local transaction and executes an RPC on a remote server with `SET REMOTE_PROC_TRANSACTIONS ON`, the local bound transaction is automatically promoted to a distributed bound transaction by MS DTC and an MS DTC session is started.

##### When to use Bound Sessions

In earlier versions of SQL Server, bound sessions were primarily used in developing extended stored procedures that must execute Transact-SQL statements on behalf of the process that calls them. Having the calling process pass in a bind token as one parameter of the extended stored procedure allows the procedure to join the transaction space of the calling process, thereby integrating the extended stored procedure with the calling process.

In the SQL Server Database Engine, stored procedures written using CLR are more secure, scalable, and stable than extended stored procedures. CLR-stored procedures use the **SqlContext** object to join the context of the calling session, not `sp_bindsession`.

Bound sessions can be used to develop three-tier applications in which business logic is incorporated into separate programs that work cooperatively on a single business transaction. These programs must be coded to carefully coordinate their access to a database. Because the two sessions share the same locks, the two programs must not try to modify the same data at the same time. At any point in time, only one session can be doing work as part of the transaction; there can be no parallel execution. The transaction can only be switched between sessions at well-defined yield points, such as when all DML statements have completed and their results have been retrieved.

### Coding efficient transactions

It is important to keep transactions as short as possible. When a transaction is started, a database management system (DBMS) must hold many resources until the end of the transaction to protect the atomicity, consistency, isolation, and durability (ACID) properties of the transaction. If data is modified, the modified rows must be protected with exclusive locks that prevent any other transaction from reading the rows, and exclusive locks must be held until the transaction is committed or rolled back. Depending on transaction isolation level settings, `SELECT` statements may acquire locks that must be held until the transaction is committed or rolled back. Especially in systems with many users, transactions must be kept as short as possible to reduce locking contention for resources between concurrent connections. Long-running, inefficient transactions may not be a problem with small numbers of users, but they are intolerable in a system with thousands of users. Beginning with SQL Server 2014 (12.x) SQL Server supports delayed durable transactions. Delayed durable transactions do not guarantee durability. See the topic Transaction Durability for more information.

#### 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. Even if users respond immediately, human reaction times are vastly slower than computer speeds. All resources held by the transaction are held for an extremely long time, which has the potential to cause blocking problems. If users do not respond, the transaction remains active, locking critical resources until they respond, which may not happen for several minutes or even hours.

- 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.
  After you know the modifications that have to be made, start a transaction, execute the modification statements, and then immediately commit or roll back. Do not open the transaction before it is required.

- To reduce blocking, consider using a row versioning-based isolation level for read-only queries.

- Make intelligent use of lower transaction isolation levels.

  Many applications can be readily coded to use a read-committed transaction isolation level. Not all transactions require the serializable transaction isolation level.

- 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.

#### Avoiding concurrency and resource problems

To prevent concurrency and resource problems, manage implicit transactions carefully. When using implicit transactions, the next Transact-SQL statement after `COMMIT` or `ROLLBACK` automatically starts a new transaction. This can cause a new transaction to be opened while the application browses through data, or even when it requires input from the user. After completing the last transaction required to protect data modifications, turn off implicit transactions until a transaction is once again required to protect data modifications. This process lets the SQL Server Database Engine use autocommit mode while the application is browsing data and getting input from the user.

In addition, when the snapshot isolation level is enabled, although a new transaction will not hold locks, a long-running transaction will prevent the old versions from being removed from `tempdb`.

### 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_TIME` option. These options control the frequency of active and indirect checkpoints, respectively. For more

information about the types of checkpoints, see Database Checkpoints (SQL Server).

- 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. For more information, see SQL Server Transaction Log Architecture and Management Guide, Troubleshoot a Full Transaction Log (SQL Server Error 9002), and The Transaction Log (SQL Server).

#### 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**).

  For more information, see sys.dm_tran_database_transactions (Transact-SQL).

- `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). For more information, see DBCC OPENTRAN (Transact-SQL).

#### Stopping a Transaction

You may have to use the KILL statement. Use this statement very carefully, however, especially when critical processes are running. For more information, see KILL (Transact-SQL).

## Additional Reading

Overhead of Row Versioning
Extended Events
sys.dm_tran_locks (Transact-SQL)
Dynamic Management Views and Functions (Transact-SQL)
Transaction Related Dynamic Management Views and Functions (Transact-SQL)

# Back Up and Restore of SQL Server Databases

5/3/2018 • 11 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database (Managed Instance only) Azure SQL Data Warehouse Parallel Data Warehouse

This topic describes the benefits of backing up SQL Server databases, basic backup and restore terms, and introduces backup and restore strategies for SQL Server and security considerations for SQL Server backup and restore.

> **IMPORTANT**
> On Azure SQL Database Managed Instance, this T-SQL feature has certain behavior changes. See Azure SQL Database Managed Instance T-SQL differences from SQL Server for details for all T-SQL behavior changes.

> **Looking for step by step instructions?** This topic does **not provide any specific steps for how to do a back up!** If you want to get right to actually backing up, scroll down this page to the links section, organized by backup tasks and whether you want to use SSMS or T-SQL.

The SQL Server backup and restore component provides an essential safeguard for protecting critical data stored in your SQL Server databases. To minimize the risk of catastrophic data loss, you need to back up your databases to preserve modifications to your data on a regular basis. A well-planned backup and restore strategy helps protect databases against data loss caused by a variety of failures. Test your strategy by restoring a set of backups and then recovering your database to prepare you to respond effectively to a disaster.

In addition to local storage for storing the backups, SQL Server also supports backup to and restore from the Windows Azure Blob Storage Service. For more information, see SQL Server Backup and Restore with Microsoft Azure Blob Storage Service. For database files stored using the Microsoft Azure Blob storage service, SQL Server 2016 (13.x) provides the option to use Azure snapshots for nearly instantaneous backups and faster restores. For more information, see File-Snapshot Backups for Database Files in Azure.

## Why back up?

- Backing up your SQL Server databases, running test restores procedures on your backups, and storing copies of backups in a safe, off-site location protects you from potentially catastrophic data loss. **Backing up is the only way to proctect your data.**

  With valid backups of a database, you can recover your data from many failures, such as:

  - Media failure.
  - User errors, for example, dropping a table by mistake.
  - Hardware failures, for example, a damaged disk drive or permanent loss of a server.
  - Natural disasters. By using SQL Server Backup to Windows Azure Blob storage service, you can create an off-site backup in a different region than your on-premises location, to use in the event of a natural disaster affecting your on-premises location.

- Additionally, backups of a database are useful for routine administrative purposes, such as copying a database from one server to another, setting up Always On availability groups or database mirroring, and archiving.

## Glossary of backup terms

**back up** [verb]
Copies the data or log records from a SQL Server database or its transaction log to a backup device, such as a disk, to create a data backup or log backup.

**backup** [noun]
A copy of data that can be used to restore and recover the data after a failure. Backups of a database can also be used to restore a copy the database to a new location.

**backup** device
A disk or tape device to which SQL Server backups are written and from which they can be restored. SQL Server backups can also be written to a Windows Azure Blob storage service, and **URL** format is used to specify the destination and the name of the backup file.. For more information, see SQL Server Backup and Restore with Microsoft Azure Blob Storage Service.

**backup media**
One or more tapes or disk files to which one or more backup have been written.

**data backup**
A backup of data in a complete database (a database backup), a partial database ( a partial backup), or a set of data files or filegroups (a file backup).

**database backup**
A backup of a database. Full database backups represent the whole database at the time the backup finished. Differential database backups contain only changes made to the database since its most recent full database backup.

**differential backup**
A data backup that is based on the latest full backup of a complete or partial database or a set of data files or filegroups (the differential base) and that contains only the data that has changed since that base.

**full backup**
A data backup that contains all the data in a specific database or set of filegroups or files, and also enough log to allow for recovering that data.

**log backup**
A backup of transaction logs that includes all log records that were not backed up in a previous log backup. (full recovery model)

**recover**
To return a database to a stable and consistent state.

**recovery**
A phase of database startup or of a restore with recovery that brings the database into a transaction-consistent state.

**recovery model**
A database property that controls transaction log maintenance on a database. Three recovery models exist: simple, full, and bulk-logged. The recovery model of database determines its backup and restore requirements.

**restore**
A multi-phase process that copies all the data and log pages from a specified SQL Server backup to a specified database, and then rolls forward all the transactions that are logged in the backup by applying logged changes to bring the data forward in time.

## Backup and restore strategies

Backing up and restoring data must be customized to a particular environment and must work with the available resources. Therefore, a reliable use of backup and restore for recovery requires a backup and restore strategy. A well-designed backup and restore strategy maximizes data availability and minimizes data loss, while considering your particular business requirements.

**Important!**

**Place the database and backups on separate devices. Otherwise, if the device containing the database fails, your backups will be unavailable. Placing the data and backups on separate devices also enhances the I/O performance for both writing backups and the production use of the database.**

A backup and restore strategy contains a backup portion and a restore portion. The backup part of the strategy defines the type and frequency of backups, the nature and speed of the hardware that is required for them, how backups are to be tested, and where and how backup media is to be stored (including security considerations). The restore part of the strategy defines who is responsible for performing restores and how restores should be performed to meet your goals for availability of the database and for minimizing data loss. We recommend that you document your backup and restore procedures and keep a copy of the documentation in your run book.

Designing an effective backup and restore strategy requires careful planning, implementation, and testing. Testing is required. You do not have a backup strategy until you have successfully restored backups in all the combinations that are included in your restore strategy. You must consider a variety of factors. These include the following:

- The production goals of your organization for the databases, especially the requirements for availability and protection of data from loss.
- The nature of each of your databases: its size, its usage patterns, the nature of its content, the requirements for its data, and so on.
- Constraints on resources, such as: hardware, personnel, space for storing backup media, the physical security of the stored media, and so on.

### Impact of the recovery model on backup and restore

Backup and restore operations occur within the context of a recovery model. A recovery model is a database property that controls how the transaction log is managed. Also, the recovery model of a database determines what types of backups and what restore scenarios are supported for the database. Typically a database uses either the simple recovery model or the full recovery model. The full recovery model can be supplemented by switching to the bulk-logged recovery model before bulk operations. For an introduction to these recovery models and how they affect transaction log management, see The Transaction Log (SQL Server)

The best choice of recovery model for the database depends on your business requirements. To avoid transaction log management and simplify backup and restore, use the simple recovery model. To minimize work-loss exposure, at the cost of administrative overhead, use the full recovery model. For information about the effect of recovery models on backup and restore, see Backup Overview (SQL Server).

### Design your backup strategy

After you have selected a recovery model that meets your business requirements for a specific database, you have to plan and implement a corresponding backup strategy. The optimal backup strategy depends on a variety of factors, of which the following are especially significant:

- How many hours a day do applications have to access the database?

  If there is a predictable off-peak period, we recommend that you schedule full database backups for that period.

- How frequently are changes and updates likely to occur?

  If changes are frequent, consider the following:

  - Under the simple recovery model, consider scheduling differential backups between full database

backups. A differential backup captures only the changes since the last full database backup.

  - Under the full recovery model, you should schedule frequent log backups. Scheduling differential backups between full backups can reduce restore time by reducing the number of log backups you have to restore after restoring the data.

- Are changes likely to occur in only a small part of the database or in a large part of the database?

  For a large database in which changes are concentrated in a part of the files or filegroups, partial backups and or file backups can be useful. For more information, see Partial Backups (SQL Server) and Full File Backups (SQL Server).

- How much disk space will a full database backup require?

  **Estimate the size of a full database backup**

  Before you implement a backup and restore strategy, you should estimate how much disk space a full database backup will use. The backup operation copies the data in the database to the backup file. The backup contains only the actual data in the database and not any unused space. Therefore, the backup is usually smaller than the database itself. You can estimate the size of a full database backup by using the **sp_spaceused** system stored procedure. For more information, see sp_spaceused (Transact-SQL).

**Schedule backups**

Performing a backup operation has minimal effect on transactions that are running; therefore, backup operations can be run during regular operations. You can perform a SQL Server backup with minimal effect on production workloads.

> For information about concurrency restrictions during backup, see Backup Overview (SQL Server).

After you decide what types of backups you require and how frequently you have to perform each type, we recommend that you schedule regular backups as part of a database maintenance plan for the database. For information about maintenance plans and how to create them for database backups and log backups, see Use the Maintenance Plan Wizard.

**Test your backups!**

You do not have a restore strategy until you have tested your backups. It is very important to thoroughly test your backup strategy for each of your databases by restoring a copy of the database onto a test system. You must test restoring every type of backup that you intend to use.

We recommend that you maintain an operations manual for each database. This operations manual should document the location of the backups, backup device names (if any), and the amount of time that is required to restore the test backups.

## More about backup tasks

- Create a Maintenance Plan
- Create a Job
- Schedule a Job

## Working with backup devices and backup media

- Define a Logical Backup Device for a Disk File (SQL Server)
- Define a Logical Backup Device for a Tape Drive (SQL Server)
- Specify a Disk or Tape As a Backup Destination (SQL Server)

- Delete a Backup Device (SQL Server)
- Set the Expiration Date on a Backup (SQL Server)
- View the Contents of a Backup Tape or File (SQL Server)
- View the Data and Log Files in a Backup Set (SQL Server)
- View the Properties and Contents of a Logical Backup Device (SQL Server)
- Restore a Backup from a Device (SQL Server)

## Creating backups

**Note!** For partial or copy-only backups, you must use the Transact-SQLBACKUP statement with the PARTIAL or COPY_ONLY option, respectively.

**Using SSMS**

- Create a Full Database Backup (SQL Server)
- Back Up a Transaction Log (SQL Server)
- Back Up Files and Filegroups (SQL Server)
- Create a Differential Database Backup (SQL Server)

**Using T-SQL**

- Use Resource Governor to Limit CPU Usage by Backup Compression (Transact-SQL)
- Back Up the Transaction Log When the Database Is Damaged (SQL Server)
- Enable or Disable Backup Checksums During Backup or Restore (SQL Server)
- Specify Whether a Backup or Restore Operation Continues or Stops After Encountering an Error (SQL Server)

## Restore data backups

**Using SSMS**

- Restore a Database Backup Using SSMS
- Restore a Database to a New Location (SQL Server)
- Restore a Differential Database Backup (SQL Server)
- Restore Files and Filegroups (SQL Server)

**Using T-SQL**

- Restore a Database Backup Under the Simple Recovery Model (Transact-SQL)
- Restore a Database to the Point of Failure Under the Full Recovery Model (Transact-SQL)
- Restore Files and Filegroups over Existing Files (SQL Server)
- Restore Files to a New Location (SQL Server)
- Restore the master Database (Transact-SQL)

## Restore transaction logs (Full Recovery Model)

**Using SSMS**

- Restore a Database to a Marked Transaction (SQL Server Management Studio)
- Restore a Transaction Log Backup (SQL Server)
- Restore a SQL Server Database to a Point in Time (Full Recovery Model)

**Using T-SQL**

- Restore a SQL Server Database to a Point in Time (Full Recovery Model)
- Restart an Interrupted Restore Operation (Transact-SQL)
- Recover a Database Without Restoring Data (Transact-SQL)

## More information and resources

Backup Overview (SQL Server)
Restore and Recovery Overview (SQL Server)
BACKUP (Transact-SQL)
RESTORE (Transact-SQL)
Backup and Restore of Analysis Services Databases
Back Up and Restore Full-Text Catalogs and Indexes
Back Up and Restore Replicated Databases
The Transaction Log (SQL Server)
Recovery Models (SQL Server)
Media Sets, Media Families, and Backup Sets (SQL Server)

# Binary Large Object (Blob) Data (SQL Server)

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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

## Compare Options for Storing Blobs in SQL Server

Compare the advantages of FILESTREAM, FileTables, and Remote Blob Store. See Compare Options for Storing Blobs (SQL Server).

## Options for Storing Blobs

### 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)

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) (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.

# Collation and Unicode Support

5/3/2018 • 16 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Collations in SQL Server provide sorting rules, case, and accent sensitivity properties for your data. Collations that are used with character data types such as **char** and **varchar** dictate the code page and corresponding characters that can be represented for that data type. Whether you are installing a new instance of SQL Server, restoring a database backup, or connecting server to client databases, it is important that you understand the locale requirements, sorting order, and case and accent sensitivity of the data that you are working with. To list the collations available on your instance of SQL Server, see sys.fn_helpcollations (Transact-SQL).

When you select a collation for your server, database, column, or expression, you are assigning certain characteristics to your data that affects the results of many operations in the database. For example, when you construct a query by using ORDER BY, the sort order of your result set might depend on the collation that is applied to the database or dictated in a COLLATE clause at the expression level of the query.

To best use collation support in SQL Server, you must understand the terms that are defined in this topic, and how they relate to the characteristics of your data.

## Collation Terms

- Collation
- Locale
- Code page
- Sort order

**Collation**

A collation specifies the bit patterns that represent each character in a data set. Collations also determine the rules that sort and compare data. SQL Server supports storing objects that have different collations in a single database. For non-Unicode columns, the collation setting specifies the code page for the data and which characters can be represented. Data that is moved between non-Unicode columns must be converted from the source code page to the destination code page.

Transact-SQL statement results can vary when the statement is run in the context of different databases that have different collation settings. If it is possible, use a standardized collation for your organization. This way, you do not have to explicitly specify the collation in every character or Unicode expression. If you must work with objects that have different collation and code page settings, code your queries to consider the rules of collation precedence. For more information, see Collation Precedence (Transact-SQL).

The options associated with a collation are case sensitivity, accent sensitivity, Kana-sensitivity, width sensitivity, variation-selector-sensitivity. These options are specified by appending them to the collation name. For example, this collation `Japanese_Bushu_Kakusu_100_CS_AS_KS_WS` is case-sensitive, accent-sensitive, Kana-sensitive, and width-sensitive. As another example, this collation `Japanese_Bushu_Kakusu_140_CI_AI_KS_WS_VSS` is case-insensitive, accent-insensitive, Kana-sensitive, width-sensitive, and variation-selector-sensitive. The following table describes the behavior associated with these various options.

| OPTION | DESCRIPTION |
|---|---|
| Case-sensitive (_CS) | Distinguishes between uppercase and lowercase letters. If selected, lowercase letters sort ahead of their uppercase versions. If this option is not selected, the collation is case-insensitive. That is, SQL Server considers the uppercase and lowercase versions of letters to be identical for sorting purposes. You can explicitly select case insensitivity by specifying _CI. |
| Accent-sensitive (_AS) | Distinguishes between accented and unaccented characters. For example, 'a' is not equal to 'ấ'. If this option is not selected, the collation is accent-insensitive. That is, SQL Server considers the accented and unaccented versions of letters to be identical for sorting purposes. You can explicitly select accent insensitivity by specifying _AI. |
| Kana-sensitive (_KS) | Distinguishes between the two types of Japanese kana characters: Hiragana and Katakana. If this option is not selected, the collation is Kana-insensitive. That is, SQL Server considers Hiragana and Katakana characters to be equal for sorting purposes. Omitting this option is the only method of specifying Kana-insensitivity. |
| Width-sensitive (_WS) | Distinguishes between full-width and half-width characters. If this option is not selected, SQL Server considers the full-width and half-width representation of the same character to be identical for sorting purposes. Omitting this option is the only method of specifying width-insensitivity. |
| Variation-selector-sensitive (_VSS) | Distinguishes between various ideographic variation selectors in Japanese collations Japanese_Bushu_Kakusu_140 and Japanese_XJIS_140 first introduced in SQL Server 2017 (14.x). A variation sequence consists of a base character plus an additional variation selector. If this _VSS option is not selected, the collation is variation selector insensitive, and the variation selector is not considered in the comparison. That is, SQL Server considers characters built upon the same base character with differing variation selectors to be identical for sorting purposes. See also Unicode Ideographic Variation Database.<br><br>Variation selector sensitive (_VSS) collations are not supported in Full-text search indexes. Full-text search indexes support only Accent-Sensitive (_AS), Kana-sensitive (_KS), and Width-sensitive (_WS) options. SQL Server XML and CLR engines do not support (_VSS) Variation selectors. |

SQL Server supports the following collation sets:

Windows collations
Windows collations define rules for storing character data that are based on an associated Windows system locale. For a Windows collation, comparison of non-Unicode data is implemented by using the same algorithm as Unicode data. The base Windows collation rules specify which alphabet or language is used when dictionary sorting is applied, and the code page that is used to store non-Unicode character data. Both Unicode and non-Unicode sorting are compatible with string comparisons in a particular version of Windows. This provides consistency across data types within SQL Server, and it also lets developers sort strings in their applications by using the same rules that are used by SQL Server. For more information, see Windows Collation Name (Transact-SQL).

Binary collations
Binary collations sort data based on the sequence of coded values that are defined by the locale and data type. They are case sensitive. A binary collation in SQL Server defines the locale and the ANSI code page that is used. This enforces a binary sort order. Because they are relatively simple, binary collations help improve application performance. For non-Unicode data types, data comparisons are based on the code points that are defined in the ANSI code page. For Unicode data types, data comparisons are based on the Unicode code points. For binary collations on Unicode data types, the locale is not considered in data sorts. For example, Latin_1_General_BIN and Japanese_BIN yield identical sorting results when they are used on Unicode data.

There are two types of binary collations in SQL Server; the older **BIN** collations and the newer **BIN2** collations. In a **BIN2** collation all characters are sorted according to their code points. In a **BIN** collation only the first character is sorted according to the code point, and remaining characters are sorted according to their byte values. (Because the Intel platform is a little endian architecture, Unicode code characters are always stored byte-swapped.)

SQL Server collations
SQL Server collations (SQL_*) provide sort order compatibility with earlier versions of SQL Server. The dictionary sorting rules for non-Unicode data are incompatible with any sorting routine that is provided by Windows operating systems. However, sorting Unicode data is compatible with a particular version of Windows sorting rules. Because SQL Server collations use different comparison rules for non-Unicode and Unicode data, you see different results for comparisons of the same data, depending on the underlying data type. For more information, see SQL Server Collation Name (Transact-SQL).

> **NOTE**
>
> When you upgrade an English-language instance of SQL Server, SQL Server collations (SQL_*) can be specified for compatibility with existing instances of SQL Server. Because the default collation for an instance of SQL Server is defined during setup, make sure that you specify collation settings carefully when the following are true:
>
> - Your application code depends on the behavior of previous SQL Server collations.
>   - You must store character data that reflects multiple languages.

Setting collations are supported at the following levels of an instance of SQL Server:

Server-level collations
The default server collation is set during SQL Server setup, and also becomes the default collation of the system databases and all user databases. Note that Unicode-only collations cannot be selected during SQL Server setup because they are not supported as server-level collations.

After a collation has been assigned to the server, you cannot change the collation except by exporting all database objects and data, rebuilding the **master** database, and importing all database objects and data. Instead of changing the default collation of an instance of SQL Server, you can specify the desired collation at the time that you create a new database or database column.

Database-level collations
When a database is created or modified, you can use the COLLATE clause of the CREATE DATABASE or ALTER DATABASE statement to specify the default database collation. If no collation is specified, the database is assigned the server collation.

You cannot change the collation of system databases except by changing the collation for the server.

The database collation is used for all metadata in the database, and is the default for all string columns, temporary objects, variable names, and any other strings used in the database. When you change the collation of a user database, there can be collation conflicts when queries in the database access temporary tables. Temporary tables are always stored in the **tempdb** system database, which uses the collation for the instance. Queries that compare character data between the user database and **tempdb** may fail if the collations cause a conflict in evaluating the character data. You can resolve this by specifying the COLLATE clause in the query. For more information, see

COLLATE (Transact-SQL).

Column-level collations
When you create or alter a table, you can specify collations for each character-string column by using the COLLATE clause. If no collation is specified, the column is assigned the default collation of the database.

Expression-level collations
Expression-level collations are set when a statement is run, and they affect the way a result set is returned. This enables ORDER BY sort results to be locale-specific. Use a COLLATE clause such as the following to implement expression-level collations:

```
SELECT name FROM customer ORDER BY name COLLATE Latin1_General_CS_AI;
```

**Locale**

A locale is a set of information that is associated with a location or a culture. This can include the name and identifier of the spoken language, the script that is used to write the language, and cultural conventions. Collations can be associated with one or more locales. For more information, see Locale IDs Assigned by Microsoft.

**Code Page**

A code page is an ordered set of characters of a given script in which a numeric index, or code point value, is associated with each character. A Windows code page is typically referred to as a *character set* or *charset*. Code pages are used to provide support for the character sets and keyboard layouts that are used by different Windows system locales.

**Sort Order**

Sort order specifies how data values are sorted. This affects the results of data comparison. Data is sorted by using collations, and it can be optimized by using indexes.

## Unicode Support

Unicode is a standard for mapping code points to characters. Because it is designed to cover all the characters of all the languages of the world, there is no need for different code pages to handle different sets of characters. If you store character data that reflects multiple languages, always use Unicode data types (**nchar**, **nvarchar**, and **ntext**) instead of the non-Unicode data types (**char**, **varchar**, and **text**).

Significant limitations are associated with non-Unicode data types. This is because a non-Unicode computer is limited to use of a single code page. You might experience performance gain by using Unicode because fewer code-page conversions are required. Unicode collations must be selected individually at the database, column, or expression level because they are not supported at the server level.

The code pages that a client uses are determined by the operating system settings. To set client code pages on the Windows operating system, use **Regional Settings** in Control Panel.

When you move data from a server to a client, your server collation might not be recognized by older client drivers. This can occur when you move data from a Unicode server to a non-Unicode client. Your best option might be to upgrade the client operating system so that the underlying system collations are updated. If the client has database client software installed, you might consider applying a service update to the database client software.

You can also try to use a different collation for the data on the server. Choose a collation that maps to a code page on the client.

To use the UTF-16 collations available in SQL Server 2017 to improve searching and sorting of some Unicode characters (Windows collations only), you can select either one of the supplementary characters (_SC) collations or one of the version 140 collations.

To evaluate issues that are related to using Unicode or non-Unicode data types, test your scenario to measure performance differences in your environment. It is a good practice to standardize the collation that is used on systems across your organization, and deploy Unicode servers and clients wherever possible.

In many situations, SQL Server interacts with other servers or clients, and your organization might use multiple data access standards between applications and server instances. SQL Server clients are one of two main types:

- **Unicode clients** that use OLE DB and Open Database Connectivity (ODBC) version 3.7 or a later version.
- **Non-Unicode clients** that use DB-Library and ODBC version 3.6 or an earlier version.

  The following table provides information about using multilingual data with various combinations of Unicode and non-Unicode servers.

| SERVER | CLIENT | BENEFITS OR LIMITATIONS |
| --- | --- | --- |
| Unicode | Unicode | Because Unicode data is used throughout the system, this scenario provides the best performance and protection from corruption of retrieved data. This is the situation with ActiveX Data Objects (ADO), OLE DB, and ODBC version 3.7 or a later version. |
| Unicode | Non-Unicode | In this scenario, especially with connections between a server that is running a newer operating system and a client that is running an older version of SQL Server, or on an older operating system, there can be limitations or errors when you move data to a client computer. Unicode data on the server tries to map to a corresponding code page on the non-Unicode client to convert the data. |
| Non-Unicode | Unicode | This is not an ideal configuration for using multilingual data. You cannot write Unicode data to the non-Unicode server. Problems are likely to occur when data is sent to servers that are outside the server's code page. |
| Non-Unicode | Non-Unicode | This is a very limiting scenario for multilingual data. You can use only a single code page. |

## Supplementary Characters

SQL Server provides data types such as **nchar** and **nvarchar** to store Unicode data. These data types encode text in a format called *UTF-16*. The Unicode Consortium allocates each character a unique codepoint, which is a value in the range 0x0000 to 0x10FFFF. The most frequently used characters have codepoint values that fit into a 16-bit word in memory and on disk, but characters with codepoint values larger than 0xFFFF require two consecutive 16-bit words. These characters are called *supplementary characters*, and the two consecutive 16-bit words are called *surrogate pairs*.

Introduced in SQL Server 2012 (11.x), a new family of supplementary character (_SC) collations can be used with the data types **nchar**, **nvarchar**, and **sql_variant**. For example: `Latin1_General_100_CI_AS_SC`, or if using a Japanese collation, `Japanese_Bushu_Kakusu_100_CI_AS_SC`.

Starting in SQL Server 2014 (12.x), all new collations automatically support supplementary characters.

If you use supplementary characters:

- Supplementary characters can be used in ordering and comparison operations in collation versions 90 or greater.
- All version 100 collations support linguistic sorting with supplementary characters.
- Supplementary characters are not supported for use in metadata, such as in names of database objects.
- Databases that use collations with supplementary characters (_SC), cannot be enabled for SQL Server Replication. This is because some of the system tables and stored procedures that are created for replication, use the legacy **ntext** data type, which does not support supplementary characters.
- The SC flag can be applied to:
  - Version 90 collations
  - Version 100 collations
- The SC flag cannot be applied to:
  - Version 80 non-versioned Windows collations
  - The BIN or BIN2 binary collations
  - The SQL* collations
  - Version 140 collations (these don't need the SC flag as they already support supplementary characters)

  The following table compares the behavior of some string functions and string operators when they use supplementary characters with and without a supplementary character-aware (SCA) collation:

| STRING FUNCTION OR OPERATOR | WITH A SUPPLEMENTARY CHARACTER-AWARE (SCA) COLLATION | WITHOUT AN SCA COLLATION |
| --- | --- | --- |
| CHARINDEX<br><br>LEN<br><br>PATINDEX | The UTF-16 surrogate pair is counted as a single codepoint. | The UTF-16 surrogate pair is counted as two codepoints. |
| LEFT<br><br>REPLACE<br><br>REVERSE<br><br>RIGHT<br><br>SUBSTRING<br><br>STUFF | These functions treat each surrogate pair as a single codepoint and work as expected. | These functions may split any surrogate pairs and lead to unexpected results. |

| STRING FUNCTION OR OPERATOR | WITH A SUPPLEMENTARY CHARACTER-AWARE (SCA) COLLATION | WITHOUT AN SCA COLLATION |
| --- | --- | --- |
| NCHAR | Returns the character corresponding to the specified Unicode codepoint value in the range 0 to 0x10FFFF. If the value specified lies in the range 0 through 0xFFFF, one character is returned. For higher values, the corresponding surrogate is returned. | A value higher than 0xFFFF returns NULL instead of the corresponding surrogate. |
| UNICODE | Returns a UTF-16 codepoint in the range 0 through 0x10FFFF. | Returns a UCS-2 codepoint in the range 0 through 0xFFFF. |
| Match One Character Wildcard<br><br>Wildcard - Character(s) Not to Match | Supplementary characters are supported for all wildcard operations. | Supplementary characters are not supported for these wildcard operations. Other wildcard operators are supported. |

# GB18030 Support

GB18030 is a separate standard used in the People's Republic of China for encoding Chinese characters. In GB18030, characters can be 1, 2, or 4 bytes in length. SQL Server provides support for GB18030-encoded characters by recognizing them when they enter the server from a client-side application and converting and storing them natively as Unicode characters. After they are stored in the server, they are treated as Unicode characters in any subsequent operations. You can use any Chinese collation, preferably the latest 100 version. All _100 level collations support linguistic sorting with GB18030 characters. If the data includes supplementary characters (surrogate pairs), you can use the SC collations available in SQL Server 2017 to improve searching and sorting.

# Complex Script Support

SQL Server can support inputting, storing, changing, and displaying complex scripts. Complex scripts include the following types:

- Scripts that include the combination of both right-to-left and left-to-right text, such as a combination of Arabic and English text.
- Scripts whose characters change shape depending on their position, or when combined with other characters, such as Arabic, Indic, and Thai characters.
- Languages such as Thai that require internal dictionaries to recognize words because there are no breaks between them.

Database applications that interact with SQL Server must use controls that support complex scripts. Standard Windows form controls that are created in managed code are complex script-enabled.

# Japanese Collations added in SQL Server 2017 (14.x)

Starting in SQL Server 2017 (14.x), two new Japanese collation families are supported, with the permutations of various options (_CS, _AS, _KS, _WS, _VSS).

To list these collations, you can query the SQL Server Database Engine:

```
SELECT Name, Description FROM fn_helpcollations()
WHERE Name LIKE 'Japanese_Bushu_Kakusu_140%' OR Name LIKE 'Japanese_XJIS_140%'
```

All of the new collations have built-in support for supplementary characters, so none of the new collations have (or need) the SC flag.

These collations are supported in Database Engine indexes, memory-optimized tables, columnstore indexes, and natively compiled modules.

## Related Tasks

| TASK | TOPIC |
| --- | --- |
| Describes how to set or change the collation of the instance of SQL Server. | Set or Change the Server Collation |
| Describes how to set or change the collation of a user database. | Set or Change the Database Collation |
| Describes how to set or change the collation of a column in the database. | Set or Change the Column Collation |
| Describes how to return collation information at the server, database, or column level. | View Collation Information |
| Describes how to write Transact-SQL statements that are more portable from one language to another, or support multiple languages more easily. | Write International Transact-SQL Statements |
| Describes how to change the language of error messages and preferences for how date, time, and currency data are used and displayed. | Set a Session Language |

## Related Content

SQL Server Best Practices Collation Change

"SQL Server Best Practices Migration to Unicode"

Unicode Consortium Web site

## See Also

Contained Database Collations
Choose a Language When Creating a Full-Text Index
sys.fn_helpcollations (Transact-SQL)

# SQL Server Configuration Manager

5/3/2018 • 4 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

> For content related to previous versions of SQL Server, see 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 (13.x)) 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 (12.x) | C:\Windows\SysWOW64\SQLServerManager12.msc |
| SQL Server 2012 (11.x) | C:\Windows\SysWOW64\SQLServerManager11.msc |

**NOTE**

Because SQL Server Configuration Manager is a snap-in for the Microsoft Management Console program and not a stand-alone program, SQL Server Configuration Manager does not appear as an application in newer versions of Windows.

- **Windows 10**:
  To open SQL Server Configuration Manager, on the **Start Page**, type SQLServerManager13.msc (for SQL Server 2016 (13.x)). For previous versions of SQL Server replace 13 with a smaller number. Clicking SQLServerManager13.msc opens the Configuration Manager. To pin the Configuration Manager to the Start Page or Task Bar, right-click SQLServerManager13.msc, and then click **Open file location**. In the Windows File Explorer, right-click SQLServerManager13.msc, and then click **Pin to Start** or **Pin to taskbar**.
  - **Windows 8**:
    To open SQL Server Configuration Manager, in the **Search** charm, under **Apps**, type **SQLServerManager<version>.msc** such as **SQLServerManager13.msc**, and then press **Enter**.

SQL Server Configuration Manager and SQL Server Management Studio use Window Management Instrumentation (WMI) to view and change some server settings. WMI provides a unified way for interfacing with the API calls that manage the registry operations requested by the SQL Server tools and to provide enhanced control and manipulation over the selected SQL services of the SQL Server Configuration Manager snap-in component. For information about configuring permissions related to WMI, see Configure WMI to Show Server Status in SQL Server Tools.

To start, stop, pause, resume, or configure services on another computer by using SQL Server Configuration Manager, see Connect to Another Computer (SQL Server Configuration Manager).

## Managing Services

Use SQL Server Configuration Manager to start, pause, resume, or stop the services, to view service properties, or to change service properties.

Use SQL Server Configuration Manager to start the Database Engine using startup parameters. For more information, see Configure Server Startup Options (SQL Server Configuration Manager).

## Changing the Accounts Used by the Services

Manage the SQL Server services using SQL Server Configuration Manager.

> **IMPORTANT**
>
> Always use SQL Server tools such as SQL Server Configuration Manager to change the account used by the SQL Server or SQL Server Agent services, or to change the password for the account. In addition to changing the account name, SQL Server Configuration Manager performs additional configuration such as setting permissions in the Windows Registry so that the new account can read the SQL Server settings. Other tools such as the Windows Services Control Manager can change the account name but do not change associated settings. If the service cannot access the SQL Server portion of the registry the service may not start properly.

As an additional benefit, passwords changed using SQL Server Configuration Manager, SMO, or WMI take affect immediately without restarting the service.

## Manage Server & Client Network Protocols

SQL Server Configuration Manager allows you to configure server and client network protocols, and connectivity options. After the correct protocols are enabled, you usually do not need to change the server network connections. However, you can use SQL Server Configuration Manager if you need to reconfigure the server connections so SQL Server listens on a particular network protocol, port, or pipe. For more information about enabling protocols, see Enable or Disable a Server Network Protocol. For information about enabling access to protocols through a firewall, see Configure the Windows Firewall to Allow SQL Server Access.

SQL Server Configuration Manager allows you to manage server and client network protocols, including the ability to force protocol encryption, view alias properties, or enable/disable a protocol.

SQL Server Configuration Manager allows you to create or remove an alias, change the order in which protocols are uses, or view properties for a server alias, including:

- Server Alias — The server alias used for the computer to which the client is connecting.
- Protocol — The network protocol used for the configuration entry.
- Connection Parameters — The parameters associated with the connection address for the network protocol configuration.

  The SQL Server Configuration Manager also allows you to view information about failover cluster instances, though Cluster Administrator should be used for some actions such as starting and stopping the services.

**Available Network Protocols**

SQL Server supports Shared Memory, TCP/IP, and Named Pipes protocols. For information about choosing a network protocols, see Configure Client Protocols. SQL Server does not support the VIA, Banyan VINES Sequenced Packet Protocol (SPP), Multiprotocol, AppleTalk, or NWLink IPX/SPX network protocols. Clients previously connecting with these protocols must select a different protocol to connect to SQL Server. You cannot use SQL Server Configuration Manager to configure the WinSock proxy. To configure the WinSock proxy, see your

ISA Server documentation.

## Related Tasks

Managing Services How-to Topics (SQL Server Configuration Manager)

Start, Stop, Pause, Resume, Restart the Database Engine, SQL Server Agent, or SQL Server Browser Service

Start, Stop, or Pause the SQL Server Agent Service

Set an Instance of SQL Server to Start Automatically (SQL Server Configuration Manager)

Prevent Automatic Startup of an Instance of SQL Server (SQL Server Configuration Manager)

# Cursors

5/3/2018 • 6 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Operations in a relational database act on a complete set of rows. For example, the set of rows returned by a SELECT statement consists of all the rows that satisfy the conditions in the WHERE clause of the statement. 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.

## Concepts

Cursor Implementations
SQL Server supports three cursor implementations.

Transact-SQL cursors
Are based on the DECLARE CURSOR syntax and are used mainly in Transact-SQL scripts, stored procedures, and triggers. Transact-SQL cursors are implemented on the server and are managed by Transact-SQL statements sent from the client to the server. They may also be contained in batches, stored procedures, or triggers.

Application programming interface (API) server cursors
Support the API cursor functions in OLE DB and ODBC. API server cursors are implemented on the server. Each time a client application calls an API cursor function, the SQL Server Native Client OLE DB provider or ODBC driver transmits the request to the server for action against the API server cursor.

Client cursors
Are implemented internally by the SQL Server Native Client ODBC driver and by the DLL that implements the ADO API. Client cursors are implemented by caching all the result set rows on the client. Each time a client application calls an API cursor function, the SQL Server Native Client ODBC driver or the ADO DLL performs the cursor operation on the result set rows cached on the client.

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. In cases where a value used to determine the location of the row within the result set is modified, such as updating a column covered by a clustered index, the modified value is visible through the cursor.

Although the database API cursor models consider a forward-only cursor to be a distinct type of cursor, SQL Server does not. SQL Server considers both forward-only and scroll as options that can be applied to static, keyset-driven, and dynamic cursors. Transact-SQL cursors support forward-only static, keyset-driven, and dynamic cursors. The database API cursor models assume that static, keyset-driven, and dynamic cursors are always scrollable. When a database API cursor attribute or property is set to forward-only, SQL Server implements this as a forward-only dynamic 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.

Transact-SQL uses the term insensitive for static cursors. Some database APIs identify them as snapshot cursors.

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.

# Requesting a Cursor

SQL Server supports two methods for requesting a cursor:

- Transact-SQL

  The Transact-SQL language supports a syntax for using cursors modeled after the ISO cursor syntax.

- Database application programming interface (API) cursor functions

SQL Server supports the cursor functionality of these database APIs:

- ADO ( Microsoft ActiveX Data Object)
- OLE DB
- ODBC (Open Database Connectivity)

An application should never mix these two methods of requesting a cursor. An application that has used the API to specify cursor behaviors should not then execute a Transact-SQL DECLARE CURSOR statement to also request a Transact-SQL cursor. An application should only execute DECLARE CURSOR if it has set all the API cursor attributes back to their defaults.

If neither a Transact-SQL nor API cursor has been requested, SQL Server defaults to returning a complete result set, known as a default result set, to the application.

## Cursor Process

Transact-SQL cursors and API cursors have different syntax, but the following general process is used with all SQL Server cursors:

1. Associate a cursor with the result set of a Transact-SQL statement, and define characteristics of the cursor, such as whether the rows in the cursor can be updated.
2. Execute the Transact-SQL statement to populate the cursor.
3. Retrieve the rows in the cursor you want to see. The operation to retrieve one row or one block of rows from a cursor is called a fetch. Performing a series of fetches to retrieve rows in either a forward or backward direction is called scrolling.
4. Optionally, perform modification operations (update or delete) on the row at the current position in the cursor.
5. Close the cursor.

## Related Content

Cursor Behaviors How Cursors Are Implemented

## See Also

DECLARE CURSOR (Transact-SQL)
Cursors (Transact-SQL)
Cursor Functions (Transact-SQL)
Cursor Stored Procedures (Transact-SQL)

# Data Collection

5/3/2018 • 6 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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.

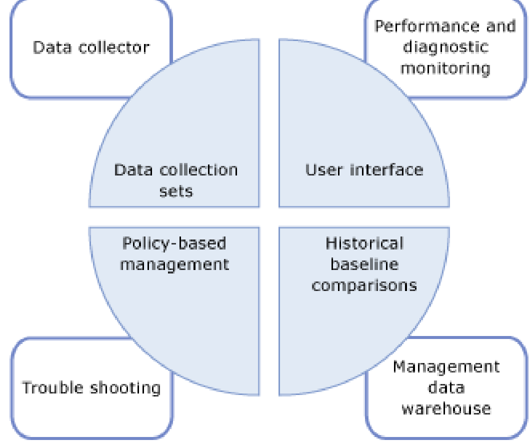
## What is Data Collector?

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 data collector enables you to adjust the scope of data collection to suit your test and production environments. The data collector also uses a data warehouse, a relational database that enables you to manage the data that you collect by setting different retention periods for your data.

The data collector supports dynamic tuning for data collection and is extensible through its API. For more information, see Data Collector Programming.

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




## Concepts

The data collector is integrated with SQL Server Agent and Integration Services, and uses both extensively. Before you work with the data collector, you should therefore understand certain concepts related to each of these SQL Server components.

SQL Server Agent is used to schedule and run collection jobs. You should understand the following concepts:

- Job
- Job step
- Job schedule

- Subsystem
- Proxy accounts

  For more information, see Automated Administration Tasks (SQL Server Agent).

  Integration Services ( SSIS) is used to execute packages that collect data from individual data providers. You should be familiar with the following SSIS tools and concepts:

- SSIS package
- SSIS package configuration

  For more information, see Integration Services (SSIS) Packages.

## Terminology

**target**
An instance of the Database Engine in an edition of SQL Server that supports Data Collection. For more information about supported editions, see the "Manageability" section of Features Supported by the Editions of SQL Server 2016.

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.

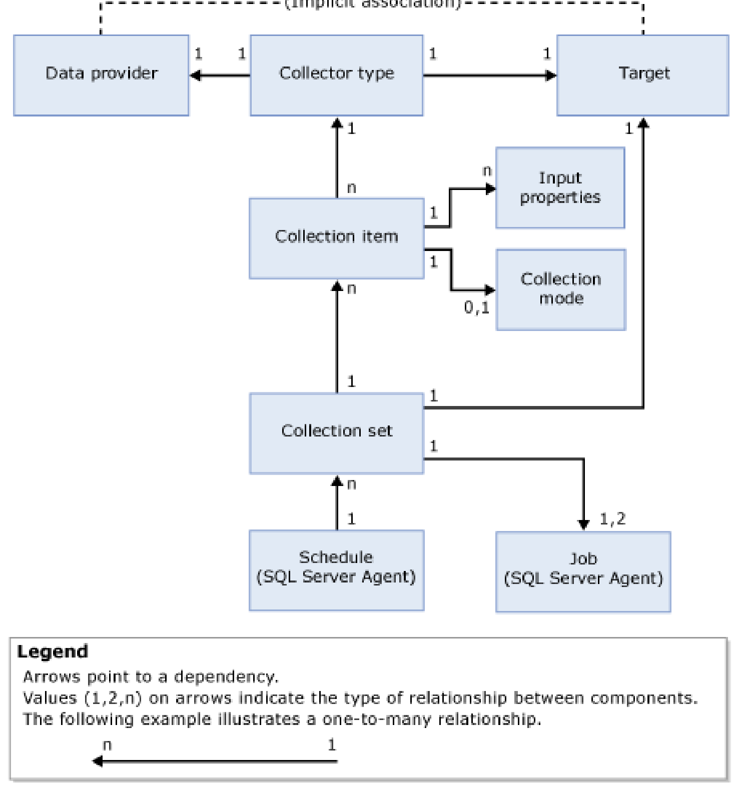
**collection item**
An instance of a collector type. A collection item is created with a specific set of input properties and a collection frequency.

**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.

## Things you can do

| DESCRIPTION | TOPIC |
| --- | --- |
| Manage different aspects of data collection, such as enabling or disabling data collection, changing a collection set configuration, or viewing data in the management data warehouse. | Manage Data Collection |
| Use reports to obtain information for monitoring system capacity and troubleshooting system performance. | System Data Collection Set Reports |
| Use the Management Data Warehouse to collect data from a server that is a data collection target. | Management Data Warehouse |
| Exploit the server-side trace capabilities of SQL Server Profiler to export a trace definition that you can use to create a collection set that uses the Generic SQL Trace collector type | Use SQL Server Profiler to Create a SQL Trace Collection Set (SQL Server Management Studio) |

# Data Compression

5/3/2018 • 12 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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.

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.

> **NOTE**
>
> Data can also be compressed using the GZIP algorithm format. This is an additional step and is most suitable for compressing portions of the data when archiving old data for long-term storage. Data compressed using the COMPRESS function cannot be indexed. For more information, see COMPRESS (Transact-SQL).

## Considerations for When You Use Row and Page Compression

When you use row and page compression, be aware the following considerations:

- The details of data compression are subject to change without notice in service packs or subsequent releases.
- Compression is available in Azure SQL Database
- Compression is not available in every edition of SQL Server. For more information, see Features Supported by the Editions of SQL Server 2016.
- Compression is not available for system tables.
- Compression can allow more rows to be stored on a page, but does not change the maximum row size of a table or index.

- A table cannot be enabled for compression when the maximum row size plus the compression overhead exceeds the maximum row size of 8060 bytes. For example, a table that has the columns c1**char(8000)** and c2**char(53)** cannot be compressed because of the additional compression overhead. When the vardecimal storage format is used, the row-size check is performed when the format is enabled. For row and page compression, the row-size check is performed when the object is initially compressed, and then checked as each row is inserted or modified. Compression enforces the following two rules:
  - An update to a fixed-length type must always succeed.
  - Disabling data compression must always succeed. Even if the compressed row fits on the page, which means that it is less than 8060 bytes; SQL Server prevents updates that would not fit on the row when it is uncompressed.
- When a list of partitions is specified, the compression type can be set to ROW, PAGE, or NONE on individual partitions. If the list of partitions is not specified, all partitions are set with the data compression property that is specified in the statement. When a table or index is created, data compression is set to NONE unless otherwise specified. When a table is modified, the existing compression is preserved unless otherwise specified.
- If you specify a list of partitions or a partition that is out of range, an error is generated.
- Nonclustered indexes do not inherit the compression property of the table. To compress indexes, you must explicitly set the compression property of the indexes. By default, the compression setting for indexes is set to NONE when the index is created.
- When a clustered index is created on a heap, the clustered index inherits the compression state of the heap unless an alternative compression state is specified.
- When a heap is configured for page-level compression, pages receive page-level compression only in the following ways:
  - Data is bulk imported with bulk optimizations enabled.
  - Data is inserted using INSERT INTO ... WITH (TABLOCK) syntax and the table does not have a nonclustered index.
  - A table is rebuilt by executing the ALTER TABLE ... REBUILD statement with the PAGE compression option.
- New pages allocated in a heap as part of DML operations do not use PAGE compression until the heap is rebuilt. Rebuild the heap by removing and reapplying compression, or by creating and removing a clustered index.
- Changing the compression setting of a heap requires all nonclustered indexes on the table to be rebuilt so that they have pointers to the new row locations in the heap.
- You can enable or disable ROW or PAGE compression online or offline. Enabling compression on a heap is single threaded for an online operation.
- The disk space requirements for enabling or disabling row or page compression are the same as for creating or rebuilding an index. For partitioned data, you can reduce the space that is required by enabling or disabling compression for one partition at a time.
- To determine the compression state of partitions in a partitioned table, query the data_compression column of the sys.partitions catalog view.
- When you are compressing indexes, leaf-level pages can be compressed with both row and page compression. Non–leaf-level pages do not receive page compression.
- Because of their size, large-value data types are sometimes stored separately from the normal row data on special purpose pages. Data compression is not available for the data that is stored separately.
- Tables that implemented the vardecimal storage format in SQL Server 2005, retain that setting when upgraded. You can apply row compression to a table that has the vardecimal storage format. However, because row compression is a superset of the vardecimal storage format, there is no reason to retain the vardecimal storage format. Decimal values gain no additional compression when you combine the vardecimal storage format with row compression. You can apply page compression to a table that has the vardecimal storage format; however, the vardecimal storage format columns probably will not achieve additional compression.

> **NOTE**
>
> SQL Server 2017 supports the vardecimal storage format; however, because row-level compression achieves the same goals, the vardecimal storage format is deprecated. This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature.

## Using Columnstore and Columnstore Archive Compression

**Applies to**: SQL Server ( SQL Server 2014 (12.x) through current version), Azure SQL Database.

### Basics

Columnstore tables and indexes are always stored with columnstore compression. You can further reduce the size of columnstore data by configuring an additional compression called archival compression. 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.

To add archival compression, use ALTER TABLE (Transact-SQL) or ALTER INDEX (Transact-SQL) with the REBUILD option and DATA COMPRESSION = COLUMNSTORE.

**Examples:**

```
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)) ;
```

To remove archival compression and restore the data to columnstore compression, use ALTER TABLE (Transact-SQL) or ALTER INDEX (Transact-SQL) with the REBUILD option and DATA COMPRESSION = COLUMNSTORE.

**Examples:**

```
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) ) ;
```

This next example sets the data compression to columnstore on some partitions, and to columnstore archival on other partitions.

```
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. For example, if you have a partition for each month of data, and most of your activity is for the most recent months, you could archive older months to reduce the storage requirements.

**Metadata**

The following system views contain information about data compression for clustered indexes:

- sys.indexes (Transact-SQL) - The **type** and **type_desc** columns include CLUSTERED COLUMNSTORE and NONCLUSTERED COLUMNSTORE.
- sys.partitions (Transact-SQL) – The **data_compression** and **data_compression_desc** columns include COLUMNSTORE and COLUMNSTORE_ARCHIVE.

The procedure sp_estimate_data_compression_savings (Transact-SQL) does not apply to columnstore indexes.

# How Compression Affects Partitioned Tables and Indexes

When you use data compression with partitioned tables and indexes, be aware of the following considerations:

- When partitions are split by using the ALTER PARTITION statement, both partitions inherit the data compression attribute of the original partition.
- When two partitions are merged, the resultant partition inherits the data compression attribute of the destination partition.
- To switch a partition, the data compression property of the partition must match the compression property of the table.
- There are two syntax variations that you can use to modify the compression of a partitioned table or index:
  - The following syntax rebuilds only the referenced partition:
    `ALTER TABLE <table_name> REBUILD PARTITION = 1 WITH (DATA_COMPRESSION = <option>)`
  - The following syntax rebuilds the whole table by using the existing compression setting for any partitions that are not referenced:

    ```
    ALTER TABLE <table_name>
    REBUILD PARTITION = ALL
    WITH (DATA_COMPRESSION = PAGE ON PARTITIONS(<range>),
    ... )
    ```

    Partitioned indexes follow the same principle using ALTER INDEX.

- When a clustered index is dropped, the corresponding heap partitions retain their data compression setting unless the partitioning scheme is modified. If the partitioning scheme is changed, all partitions are rebuilt to an uncompressed state. To drop a clustered index and change the partitioning scheme requires the following steps:

  1. Drop the clustered index.
  2. Modify the table by using the ALTER TABLE ... REBUILD ... option that specifies the compression option.

  To drop a clustered index OFFLINE is a very fast operation, because only the upper levels of clustered indexes are removed. When a clustered index is dropped ONLINE, SQL Server must rebuild the heap

two times, once for step 1 and once for step 2.

## How Compression Affects Replication

**Applies to**: SQL Server ( SQL Server 2014 (12.x) through current version).

When you are using data compression with replication, be aware of the following considerations:

- When the Snapshot Agent generates the initial schema script, the new schema uses the same compression settings for both the table and its indexes. Compression cannot be enabled on just the table and not the index.
- For transactional replication the article schema option determines what dependent objects and properties have to be scripted. For more information, see sp_addarticle.
  The Distribution Agent does not check for down-level Subscribers when it applies scripts. If the replication of compression is selected, creating the table on down-level Subscribers fails. In the case of a mixed topology, do not enable the replication of compression.
- For merge replication, publication compatibility level overrides the schema options and determines the schema objects that are scripted.
  In the case of a mixed topology, if it is not required to support the new compression options, the publication compatibility level should be set to the down-level Subscriber version. If it is required, compress tables on the Subscriber after they have been created.

The following table shows replication settings that control compression during replication.

| USER INTENT | REPLICATE PARTITION SCHEME FOR A TABLE OR INDEX | REPLICATE COMPRESSION SETTINGS | SCRIPTING BEHAVIOR |
| --- | --- | --- | --- |
| To replicate the partition scheme and enable compression on the Subscriber on the partition. | True | True | Scripts both the partition scheme and the compression settings. |
| To replicate the partition scheme but not compress the data on the Subscriber. | True | False | Scripts out the partition scheme but not the compression settings for the partition. |
| To not replicate the partition scheme and not compress the data on the Subscriber. | False | False | Does not script partition or compression settings. |
| To compress the table on the Subscriber if all the partitions are compressed on the Publisher, but not replicate the partition scheme. | False | True | Checks if all the partitions are enabled for compression.<br><br>Scripts out compression at the table level. |

## How Compression Affects Other SQL Server Components

**Applies to**: SQL Server ( SQL Server 2014 (12.x) through current version).

Compression occurs in the storage engine and the data is presented to most of the other components of SQL Server in an uncompressed state. This limits the effects of compression on the other components to the following:

- Bulk import and export operations
  When data is exported, even in native format, the data is output in the uncompressed row format. This can cause the size of exported data file to be significantly larger than the source data.
  When data is imported, if the target table has been enabled for compression, the data is converted by the

storage engine into compressed row format. This can cause increased CPU usage compared to when data is imported into an uncompressed table.
When data is bulk imported into a heap with page compression, the bulk import operation tries to compress the data with page compression when the data is inserted.

- Compression does not affect backup and restore.
- Compression does not affect log shipping.
- Data compression is incompatible with sparse columns. Therefore, tables containing sparse columns cannot be compressed nor can sparse columns be added to a compressed table.
- Enabling compression can cause query plans to change because the data is stored using a different number of pages and number of rows per page.

## See Also

Row Compression Implementation
Page Compression Implementation
Unicode Compression Implementation
CREATE PARTITION SCHEME (Transact-SQL)
CREATE PARTITION FUNCTION (Transact-SQL)
CREATE TABLE (Transact-SQL)
ALTER TABLE (Transact-SQL)
CREATE INDEX (Transact-SQL)
ALTER INDEX (Transact-SQL)

# Data-tier Applications

5/3/2018 • 8 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

A data-tier application (DAC) is a logical database management entity that defines all of the SQL Server objects - like tables, views, and instance objects, including logins – associated with a user's database. A DAC is a self-contained unit of SQL Server database deployment that enables data-tier developers and database administrators to package SQL Server objects into a portable artifact called a DAC package, also known as a DACPAC.

A BACPAC is a related artifact that encapsulates the database schema as well as the data stored in the database.

## Benefits of Data-tier Applications

The lifecycle of most database applications involves developers and DBAs sharing and exchanging scripts and ad hoc integration notes for application update and maintenance activities. While this is acceptable for a small number of databases, it quickly becomes unscalable once databases grow in number, size, and complexity.

A DAC is a database lifecycle management and productivity tool that enables declarative database development to simplify deployment and management. A developer can author a database in SQL Server Data Tool database project and then build the database into a DACPAC for handoff to a DBA. The DBA can deploy the DAC using SQL Server Management Studio to a test or production instance of SQL Server or Azure SQL Database. Alternatively, the DBA can use the DACPAC to upgrade a previously deployed database using SQL Server Management Studio. To complete the lifecycle, the DBA can extract the database into a DACPAC and hand it off to a developer to either reflect test or production adjustments, or to enable further database design changes in response to changes in the application.

The advantage of a DAC-driven deployment over a script driven exercise is that the tool helps the DBA with identifying and validating behaviors from different source and target databases. During upgrades, the tool warns the DBA if the upgrade might cause data loss, and also provide an upgrade plan. The DBA can evaluate the plan and then utilize the tool to proceed with the upgrade.

DAC's also support versioning to help the developer and the DBA maintain and manage the database lineage through its lifecycle.

## DAC Concepts

A DAC simplifies the development, deployment, and management of data-tier elements that support an application:

- A data-tier application (DAC) is a logical database management entity that defines all SQL Server objects - such as tables, views, and instance objects – associated with a user's database. It is a self-contained unit of SQL Server database deployment that enables data-tier developers and DBAs to package SQL Server objects into a portable artifact called a DAC package, or .dacpac file.
- For a SQL Server database to be treated as a DAC, it must be registered – either explicitly by a user operation, or implicitly by one of the DAC operations. When a database is registered, the DAC version and other properties are recorded as part of the metadata of the database. Conversely, a database can also be unregistered and have its DAC properties removed.
- In general, DAC tools are capable of reading DACPAC files generated by DAC tools from previous SQL Server versions, and can also deploy DACPAC's to previous versions of SQL Server. However, DAC tools

from earlier versions cannot read DACPAC files generated by DAC tools from later versions. Specifically:

- DAC operations were introduced in SQL Server 2008 R2. In addition to SQL Server 2008 R2 databases, the tools support generation of DACPAC files from SQL Server 2008, SQL Server 2005 and SQL Server 2000 databases.
- In addition to SQL 2016 databases, the tools shipped with SQL Server 2016 can read DACPAC files generated by DAC tools shipped with SQL Server 2008 R2 or SQL Server 2012. This includes databases from SQL Server 2014, 2012, 2008 R2, 2008, and 2005, but **not** SQL Server 2000.
- DAC tools from SQL Server 2008 R2 cannot read DACPAC files generated by tools from SQL Server 2012 (11.x) or SQL Server 2017.

- A DACPAC is a Windows file with a .dacpac extension. The file supports an open format consisting of multiple XML sections representing details of the DACPAC origin, the objects in the database, and other characteristics. An advanced user can unpack the file using the DacUnpack.exe utility that ships with the product to inspect each section more closely.
- The user must be a member of the dbmanager role or assigned CREATE DATABASE permissions to create a database, including creating a database by deploying a DAC package. The user must be a member of the dbmanager role, or have been assigned DROP DATABASE permissions to drop a database.

## DAC Tools

A DACPAC can be seamlessly used across multiple tools that ship with SQL Server 2017. These tools address the requirements of different user personas using a DACPAC as the unit of interoperability.

- Application Developers:
  - Can use a SQL Server Data Tools database project to design a database. A successful build of this project results in the generation of a DACPAC contained in a .dacpac file.
  - Can import a DACPAC into a database project and continue to design the database.

    SQL Server Data Tools also supports a Local DB for unconnected, client-side database application development. The developer can take a snapshot of this local database to create DACPAC contained in a .dacpac file.
  - Independently, the developer can publish a database project directly to a database without even generating a DACPAC. The publish operation follows similar behavior as the deploy operation from other tools.
- Database Administrators:
  - Can use SQL Server Management Studio to extract a DACPAC from an existing database, and also perform other DAC operations.
  - In addition, the DBA for a SQL Database can use the Management Portal for SQL Azure for DAC operations.
- Independent Software Vendors:
  - Hosting services and other data management products for SQL Server can use the DACFx API for DAC operations.
- IT Administrators:
  - IT systems integrators and administrators can use the SqlPackage.exe command line tool for DAC operations.

## DAC Operations

A DAC supports the following operations:

- **EXTRACT** – the user can extract a database into a DACPAC.
- **DEPLOY** – the user can deploy a DACPAC to a host server. When the deployment is done from a manageability tool like SQL Server Management Studio or the Management Portal for SQL Azure, the resulting database in the host server is implicitly registered as a data-tier application.
- **REGISTER** – the user can register a database as a data-tier application.
- **UNREGISTER** – a database previously registered as a DAC can be unregistered.
- **UPGRADE** – a database can be upgraded using a DACPAC. Upgrade is supported even on databases that are not previously registered as data-tier applications, but as a consequence of the upgrade, the database will be implicitly registered.

## BACPAC

A BACPAC is a Windows file with a .bacpac extension that encapsulates a database's schema and data. The primary use case for a BACPAC is to move a database from one server to another - or to migrate a database from a local server to the cloud - and archiving an existing database in an open format.

Similar to the DACPAC, the BACPAC file format is open – the schema contents of the BACPAC are identical to that of the DACPAC. The data in a BACPAC is stored in JSON format.

DACPAC and BACPAC are similar but they target different scenarios. A DACPAC is focused on capturing and deploying schema, including upgrading an existing database. The primary use case for a DACPAC is to deploy a tightly defined schema to development, test, and then to production environments. And also the reverse: capturing production's schema and applying it back to test and development environments.

A BACPAC, on the other hand, is focused on capturing schema and data supporting two main operations:

- **EXPORT**– The user can export the schema and the data of a database to a BACPAC.
- **IMPORT** – The user can import the schema and the data into a new database in the host server.

  Both these capabilities are supported by the database management tools: SQL Server Management Studio, the Azure Portal, and the DACFx API.

## Permissions

You must be a member of the **dbmanager** role or assigned **CREATE DATABASE** permissions to create a database, including creating a database by deploying a DAC package. You must be a member of the **dbmanager** role, or have been assigned **DROP DATABASE** permissions to drop a database.

## Data-tier Application Tasks

| TASK | TOPIC LINK |
| --- | --- |
| Describes how to use a DAC package file to create a new DAC instance. | Deploy a Data-tier Application |
| Describes how to use a new DAC package file to upgrade an instance to a new version of the DAC. | Upgrade a Data-tier Application |

| TASK | TOPIC LINK |
| --- | --- |
| Describes how to remove a DAC instance. You can choose to also detach or drop the associated database, or leave the database intact. | Delete a Data-tier Application |
| Describes how to view the health of currently deployed DACs by using the SQL Server Utility. | Monitor Data-tier Applications |
| Describes how to create a .bacpac file that contains an archive of the data and metadata in a DAC. | Export a Data-tier Application |
| Describes how to use a DAC archive file (.bacpac) to either perform a logical restore of a DAC, or to migrate the DAC to another instance of the Database Engine or SQL Database. | Import a BACPAC File to Create a New User Database |
| Describes how to import a BACPAC file to create a new user database within an instance of SQL Server. | Extract a DAC From a Database |
| Describes how to promote an existing database to be a DAC instance. A DAC definition is built and stored in the system databases. | Register a Database As a DAC |
| Describes how to review the contents of a DAC package and the actions a DAC upgrade will perform before using the package in a production system. | Validate a DAC Package |
| Describes how to place the contents of a DAC package into a folder where a database administrator can review what the DAC does before deploying it to a production server. | Unpack a DAC Package |
| Describes how to use a wizard to deploy an existing database. The wizard uses DACs to perform the deployment. | Deploy a Database By Using a DAC |

## See also

DAC Support For SQL Server Objects and Versions

# Database Mail

5/3/2018 • 5 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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**

- Off by default: To reduce the surface area of SQL Server, Database Mail stored procedures are disabled by default.
- 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 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.

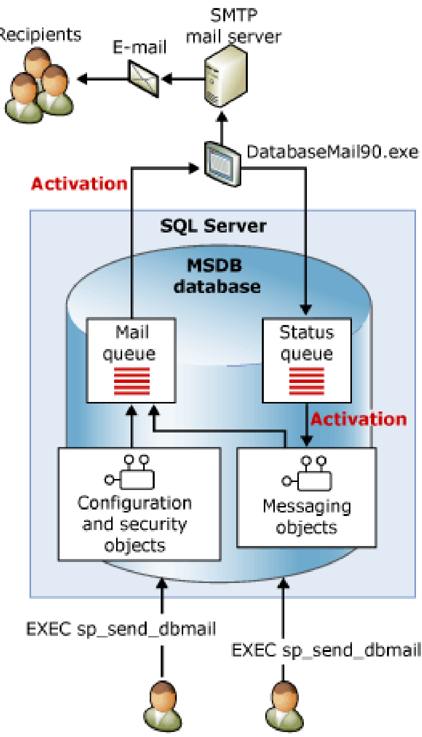
- Logging. Database Mail logs e-mail activity to SQL Server, the Microsoft Windows application event log, and to tables in the **msdb** database.

- 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). The external process reads the e-mail information and sends the e-mail message to the appropriate e-mail server or servers. The external process inserts an item in the Status queue for the outcome of the send operation. Inserting the new entry in the status queue starts an internal stored procedure that updates the status of the e-mail message. Besides storing the sent, or unsent, e-mail message, Database Mail also records any e-mail attachments in the system tables. Database Mail views provide the status of messages for troubleshooting, and stored procedures allow for administration of the Database Mail queue.




## Introduction to Database Mail Components

Database Mail consists of the following main components:

- Configuration and security components

  Database Mail stores configuration and security information in the **msdb** database. Configuration and security objects create profiles and accounts used by Database Mail.

- Messaging components

  The **msdb** database acts as the mail-host database that holds the messaging objects that Database Mail uses to send e-mail. These objects include the **sp_send_dbmail** stored procedure and the data structures that hold information about messages.

- Database Mail executable

  The Database Mail executable is an external program that reads from a queue in the **msdb** database and sends messages to e-mail servers.

- Logging and auditing components

  Database Mail records logging information in the **msdb** database and the Microsoft Windows application event log.

  **Configuring Agent to use Database Mail:**

  SQL Server Agent can be configured to use Database Mail. This is required for alert notifications and automatic notification when a job completes.

> **WARNING**
> Individual job steps within a job can also send e-mail without configuring SQL Server Agent to use Database Mail. For example, a Transact-SQL job step can use Database Mail to send the results of a query to a list of recipients.

You can configure SQL Server Agent to send e-mail messages to predefined operators when:

- An alert is triggered. Alerts can be configured to send e-mail notification of specific events that occur. For example, alerts can be configured to notify an operator of a particular database event or operating system condition that may need immediate action. For more information about configuring alerts, see Alerts.

- A scheduled task, such as a database backup or replication event, succeeds or fails. For example, you can use SQL Server Agent Mail to notify operators if an error occurs during processing at the end of a month.

## Database Mail Component Topics

- Database Mail Configuration Objects

- Database Mail Messaging Objects

- Database Mail External Program

- Database Mail Log and Audits

- Configure SQL Server Agent Mail to Use Database Mail

# Databases

5/3/2018 • 2 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

A database in SQL Server is made up of a collection of tables that stores a specific set of structured data. A table contains a collection of rows, also referred to as records or tuples, and columns, also referred to as attributes. Each column in the table is designed to store a certain type of information, for example, dates, names, dollar amounts, and numbers.

## Basic Information about Databases

A computer can have one or more than one instance of SQL Server installed. Each instance of SQL Server can contain one or many databases. Within a database, there are one or many object ownership groups called schemas. Within each schema there are database objects such as tables, views, and stored procedures. Some objects such as certificates and asymmetric keys are contained within the database, but are not contained within a schema. For more information about creating tables, see Tables.

SQL Server databases are stored in the file system in files. Files can be grouped into filegroups. For more information about files and filegroups, see Database Files and Filegroups.

When people gain access to an instance of SQL Server they are identified as a login. When people gain access to a database they are identified as a database user. A database user can be based on a login. If contained databases are enabled, a database user can be created that is not based on a login. For more information about users, see CREATE USER (Transact-SQL).

A user that has access to a database can be given permission to access the objects in the database. Though permissions can be granted to individual users, we recommend creating database roles, adding the database users to the roles, and then grant access permission to the roles. Granting permissions to roles instead of users makes it easier to keep permissions consistent and understandable as the number of users grow and continually change. For more information about roles permissions, see CREATE ROLE (Transact-SQL) and Principals (Database Engine).

## Working with Databases

Most people who work with databases use the SQL Server Management Studio tool. The Management Studio tool has a graphical user interface for creating databases and the objects in the databases. Management Studio also has a query editor for interacting with databases by writing Transact-SQL statements. Management Studio can be installed from the SQL Server installation disk, or downloaded from MSDN.

## In This Section

| | |
|---|---|
| System Databases | Delete Data or Log Files from a Database |
| Contained Databases | Display Data and Log Space Information for a Database |
| SQL Server Data Files in Microsoft Azure | Increase the Size of a Database |

| | |
|---|---|
| Database Files and Filegroups | Rename a Database |
| Database States | Set a Database to Single-user Mode |
| File States | Shrink a Database |
| Estimate the Size of a Database | Shrink a File |
| Copy Databases to Other Servers | View or Change the Properties of a Database |
| Database Detach and Attach (SQL Server) | View a List of Databases on an Instance of SQL Server |
| Add Data or Log Files to a Database | View or Change the Compatibility Level of a Database |
| Change the Configuration Settings for a Database | Use the Maintenance Plan Wizard |
| Create a Database | Create a User-Defined Data Type Alias |
| Delete a Database | Database Snapshots (SQL Server) |

## Related Content

Indexes

Views

Stored Procedures (Database Engine)

# Common Language Runtime (CLR) Integration Programming Concepts

5/3/2018 • 2 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Beginning with SQL Server 2005, SQL Server features the integration of the common language runtime (CLR) component of the .NET Framework for Microsoft Windows. This means that you can now write stored procedures, triggers, user-defined types, user-defined functions, user-defined aggregates, and streaming table-valued functions, using any .NET Framework language, including Microsoft Visual Basic .NET and Microsoft Visual C#.

The Microsoft.SqlServer.Server namespace includes core functionality for CLR programming in SQL Server. However, the Microsoft.SqlServer.Server namespace is documented in the .NET Framework SDK. This documentation is not included in SQL Server Books Online.

> **IMPORTANT**
>
> By default, the .NET Framework is installed with SQL Server, but the .NET Framework SDK is not. Without the SDK installed on your computer and included in the Books Online collection, links to SDK content in this section do not work. Install the .NET Framework SDK. Once installed, add the SDK to the Books Online collection and table of contents by following the instructions in Installing the .NET Framework SDK.

> **NOTE**
>
> CLR functionality, such as CLR user functions, are *not* supported for Azure SQL Database.

The following table lists the topics in this section.

Common Language Runtime (CLR) Integration Overview
Provides a brief overview of the CLR, and describes how and why this technology has been used in SQL Server. Describes the benefits of using the CLR to create database objects.

Assemblies (Database Engine)
Describes how assemblies are used in SQL Server to deploy functions, stored procedures, triggers, user-defined aggregates, and user-defined types that are written in one of the managed code languages hosted by the Microsoft .NET Framework common language runtime (CLR), and not written in Transact-SQL.

Building Database Objects with Common Language Runtime (CLR) Integration
Describes the kinds of objects that can be built using the CLR, and reviews the requirements for building CLR database objects.

Data Access from CLR Database Objects
Describes how a CLR routine can access data stored in an instance of SQL Server.

CLR Integration Security
Describes the CLR integration security model.

Debugging CLR Database Objects
Describes limitations of and requirements for debugging CLR database objects.

Deploying CLR Database Objects
Describes deploying assemblies to production servers.

Managing CLR Integration Assemblies
Describes how to create and drop CLR integration assemblies.

Monitoring and Troubleshooting Managed Database Objects
Provides information about the tools that can be used to monitor and troubleshoot managed database objects and assemblies running in SQL Server.

Usage Scenarios and Examples for Common Language Runtime (CLR) Integration
Describes usage scenarios and code samples using CLR objects.

## See Also

Assemblies (Database Engine)
Installing the .NET Framework SDK

# Database Engine Extended Stored Procedures - Programming

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

**IMPORTANT**

This feature will be removed in a future version of Microsoft SQL Server. Do not use this feature in new development work, and modify applications that currently use this feature as soon as possible. Use CLR integration instead.

In the past, Open Data Services was used to write server applications, such as gateways to non-SQL Server database environments. Microsoft SQL Server does not support the obsolete portions of the Open Data Services API. The only part of the original Open Data Services API still supported by SQL Server are the extended stored procedure functions, so the API has been renamed to the Extended Stored Procedure API.

With the emergence of newer and more powerful technologies, such as distributed queries and CLR Integration, the need for Extended Stored Procedure API applications has largely been replaced.

**NOTE**

If you have existing gateway applications, you cannot use the opends60.dll that ships with SQL Server to run the applications. Gateway applications are no longer supported.

## Extended Stored Procedures vs. CLR Integration

In earlier releases of SQL Server, extended stored procedures (XPs) provided the only mechanism that was available for database application developers to write server-side logic that was either hard to express or impossible to write in Transact-SQL. CLR Integration provides a more robust alternative to writing such stored procedures. Furthermore, with CLR Integration, logic that used to be written in the form of stored procedures is often better expressed as table-valued functions, which allow the results constructed by the function to be queried in SELECT statements by embedding them in the FROM clause.

## See Also

Common Language Runtime (CLR) Integration Overview
CLR Table-Valued Functions

# Database Engine Extended Stored Procedures - Reference

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

> **IMPORTANT**
> This feature will be removed in a future version of Microsoft SQL Server. Do not use this feature in new development work, and modify applications that currently use this feature as soon as possible. Use CLR integration instead.

The Microsoft Extended Stored Procedure API, previously part of Open Data Services, provides a server-based application programming interface (API) for extending Microsoft SQL Server functionality. The API consists of C and C++ functions and macros used to build applications.

With the emergence of newer and more powerful technologies such as CLR integration, the need for extended stored procedures has largely been replaced.

> **IMPORTANT**
> You should thoroughly review the source code of extended stored procedures, and you should test the compiled DLLs before you install them on a production server. For information about security review and testing, see this Microsoft Web site.

## In This Section

| | |
|---|---|
| Data Types | srv_pfield |
| srv_alloc | |
| srv_convert | srv_pfieldex |
| srv_describe | srv_rpcdb |
| srv_getbindtoken | srv_rpcname |
| srv_got_attention | srv_rpcnumber |
| | srv_rpcoptions |
| srv_message_handler | srv_rpcowner |
| srv_paramdata | srv_rpcparams |
| srv_paraminfo | srv_senddone |

| | |
|---|---|
| srv_paramlen | srv_sendmsg |
| srv_parammaxlen | srv_sendrow |
| srv_paramname | srv_setcoldata |
| srv_paramnumber | srv_setcollen |
| srv_paramset | srv_setutype |
| srv_paramsetoutput | srv_willconvert |
| srv_paramstatus | srv_wsendmsg |
| srv_paramtype | |

# SQL Server Express LocalDB Reference - Error Messages

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

This section provides reference information about the following SQL Server Express LocalDB error messages:

LOCALDB_ERROR_ADMIN_RIGHTS_REQUIRED
Administrator privileges are required in order to execute this operation.

LOCALDB_ERROR_AUTO_INSTANCE_CREATE_FAILED
Cannot create an automatic instance. See the Windows Application event log for error details.

LOCALDB_ERROR_CALLER_IS_NOT_OWNER
API caller is not LocalDB instance owner.

LOCALDB_ERROR_CANNOT_ACCESS_INSTANCE_FOLDER
An instance folder cannot be accessed.

LOCALDB_ERROR_CANNOT_ACCESS_INSTANCE_REGISTRY
An instance registry cannot be accessed.

LOCALDB_ERROR_CANNOT_CREATE_INSTANCE_FOLDER
A folder cannot be created under %userprofile%.

LOCALDB_ERROR_CANNOT_CREATE_SQL_PROCESS
A process for SQL Server cannot be created.

LOCALDB_ERROR_CANNOT_GET_USER_PROFILE_FOLDER
A user profile folder cannot be retrieved.

LOCALDB_ERROR_CANNOT_MODIFY_INSTANCE_REGISTRY
An instance registry cannot be modified.

LOCALDB_ERROR_INSTANCE_ALREADY_SHARED
The specified instance is already shared.

LOCALDB_ERROR_INSTANCE_BUSY
The specified instance is running.

LOCALDB_ERROR_INSTANCE_CONFIGURATION_CORRUPT
An instance configuration is corrupted.

LOCALDB_ERROR_INSTANCE_EXISTS_WITH_LOWER_VERSION
The specified instance already exists but its version is lower than requested.

LOCALDB_ERROR_INSTANCE_FOLDER_PATH_TOO_LONG
The path where the instance should be stored is longer than MAX_PATH.

LOCALDB_ERROR_INSTANCE_STOP_FAILED
The stop operation failed to complete within the given time.

LOCALDB_ERROR_INSUFFICIENT_BUFFER

The input buffer is too short, and truncation was not requested.

LOCALDB_ERROR_INTERNAL_ERROR
An unexpected error occurred. See the event log for details.

LOCALDB_ERROR_INVALID_INSTANCE_NAME
The specified instance name is invalid.

LOCALDB_ERROR_INVALID_PARAMETER
One or more specified input parameters are invalid.

LOCALDB_ERROR_NOT_INSTALLED
SQL Server Express LocalDB is not installed on the computer.

LOCALDB_ERROR_SHARED_NAME_TAKEN
The specified shared name is already taken.

LOCALDB_ERROR_SQL_SERVER_STARTUP_FAILED
A SQL Server process was started, but SQL Server startup failed.

LOCALDB_ERROR_TOO_MANY_SHARED_INSTANCES
There are too many shared instances.

LOCALDB_ERROR_UNKNOWN_ERROR_CODE
The requested message does not exist.

LOCALDB_ERROR_UNKNOWN_INSTANCE
The instance does not exist.

LOCALDB_ERROR_UNKNOWN_LANGUAGE_ID
The message is not available in the requested language.

LOCALDB_ERROR_UNKNOWN_VERSION
The specified version is not available.

LOCALDB_ERROR_VERSION_REQUESTED_NOT_INSTALLED
The specified patch level is not installed.

LOCALDB_ERROR_WAIT_TIMEOUT
A time-out occurred while trying to acquire the synchronization locks.

LOCALDB_ERROR_XEVENT_FAILED
Failed to start XEvent engine within the LocalDB Instance API.

# SQL Server Express LocalDB Reference - Instance APIs

5/3/2018 • 5 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

In the traditional, service-based SQL Server world, individual SQL Server instances installed on a single computer are physically separated; that is, each instance must be installed and removed separately, has a separate set of binaries, and runs under a separate service process. The SQL Server instance name is used to specify which SQL Server instance the user wants to connect to.

The SQL Server Express LocalDB instance API uses a simplified, "light" instance model. Although individual LocalDB instances are separated on the disk and in the registry, they use the same set of shared LocalDB binaries. Moreover, LocalDB does not use services; LocalDB instances are launched on demand through LocalDB instance API calls. In LocalDB, the instance name is used to specify which of the LocalDB instances the user wants to work with.

A LocalDB instance is always owned by a single user and is visible and accessible only from this user's context, unless instance sharing is enabled.

Although technically LocalDB instances are not the same as traditional SQL Server instances, their intended use is similar. They are called *instances* to emphasize this similarity and to make them more intuitive to SQL Server users.

LocalDB supports two kinds of instances: automatic instances (AI) and named instances (NI). The identifier for a LocalDB instance is the instance name.

## Automatic LocalDB Instances

Automatic LocalDB instances are "public"; they are created and managed automatically for the user and can be used by any application. One automatic LocalDB instance exists for every version of LocalDB that is installed on the user's computer.

Automatic LocalDB instances provide seamless instance management. The user does not need to create the instance. This enables users to easily install applications and to migrate to different computers. If the target computer has the specified version of LocalDB installed, the automatic LocalDB instance for that version is also available on that computer.

#### Automatic Instance Management

A user does not need to create an automatic LocalDB instance. The instance is lazily created the first time the instance is used, provided that the specified version of LocalDB is available on the user's computer. From the user's point of view, the automatic instance is always present if the LocalDB binaries are present.

Other instance management operations, such as Delete, Share, and Unshare, also work for automatic instances. In particular, deleting an automatic instance effectively resets the instance, which will be re-created on the next Start operation. Deleting an automatic instance may be required if the system databases become corrupted.

#### Automatic Instance Naming Rules

Automatic LocalDB instances have a special pattern for the instance name that belongs to a reserved namespace. This is necessary to prevent name conflicts with named LocalDB instances.

The automatic instance name is the LocalDB baseline release version number preceded by a single "v" character.

This looks like “v” plus two numbers with a period between them; for example, v11.0 or V12.00.

Examples of illegal automatic instance names are:

- 11.0 (missing the “v” character at the beginning)
- v11 (missing a period and the second number of the version)
- v11. (missing the second number of the version)
- v11.0.1.2 (version number has more than two parts)

# Named LocalDB Instances

Named LocalDB instances are “private”; an instance is owned by a single application that is responsible for creating and managing the instance. Named LocalDB instances provide isolation and improve performance.

**Named Instance Creation**

The user must create named instances explicitly through the LocalDB management API, or implicitly through the app.config file for a managed application. A managed application may also use the API.

Each named instance has an associated LocalDB version; that is, it points to a specified set of LocalDB binaries. The version for the named instance is set during the instance creation process.

**Named Instance Naming Rules**

A LocalDB instance name can have up to a total of 128 characters (the limit is imposed by the **sysname** data type). This is a significant difference compared to traditional SQL Server instance names, which are limited to NetBIOS names of 16 ASCII characters. The reason for this difference is that LocalDB treats databases as files, and therefore implies file-based semantics, so it’s intuitive for users to have more freedom in choosing instance names.

A LocalDB instance name can contain any Unicode characters that are legal within the filename component. Illegal characters in a filename component generally include the following characters: ASCII/Unicode characters 1 through 31, as well as quote ("), less than (<), greater than (>), pipe (|), backspace (\b), tab (\t), colon (:), asterisk (*), question mark (?), backslash (\), and forward slash (/). Note that the null character (\0) is allowed because it is used for string termination; everything after the first null character will be ignored.

> **NOTE**
> The list of illegal characters may depend on the operating system and may change in future releases.

Leading and trailing white spaces in instance names are ignored and will be trimmed.

To avoid naming conflicts, named LocalDB instances cannot have a name that follows the naming pattern for automatic instances, as described earlier in “Automatic Instance Naming Rules.” An attempt to create a named instance with a name that follows the automatic instance naming pattern effectively creates a default instance.

# SQL Server Express LocalDB Reference Topics

SQL Server Express LocalDB Header and Version Information
Provides header file information and registry keys for finding the LocalDB instance API.

Command-Line Management Tool: SqlLocalDB.exe
Describes SqlLocalDB.exe, a tool for managing LocalDB instances from the command line.

LocalDBCreateInstance Function
Describes the function to create a new LocalDB instance.

LocalDBDeleteInstance Function

Describes the function to remove a LocalDB instance.

LocalDBFormatMessage Function
Describes the function to return the localized description for a LocalDB error.

LocalDBGetInstanceInfo Function
Describes the function to get information for a LocalDB instance, such as whether it exists, version information, whether it is running, and so on.

LocalDBGetInstances Function
Describes the function to return all the LocalDB instances with a specified version.

LocalDBGetVersionInfo Function
Describes the function to return information for a specified LocalDB version.

LocalDBGetVersions Function
Describes the function to return all LocalDB versions available on a computer.

LocalDBShareInstance Function
Describes the function to share a specified LocalDB instance.

LocalDBStartInstance Function
Describes the function to start a specified LocalDB instance.

LocalDBStartTracing Function
Describes the function to enable API tracing for a user.

LocalDBStopInstance Function
Describes the function to stop a specified LocalDB instance from running.

LocalDBStopTracing Function
Describes the function to disable API tracing for a user.

LocalDBUnshareInstance Function
Describes the function to stop sharing a specified LocalDB instance.

# SQL Server Native Client Programming

5/3/2018 • 3 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

SQL Server Native Client is a stand-alone data access application programming interface (API), used for both OLE DB and ODBC, that was introduced in SQL Server 2005. SQL Server Native Client combines the SQL OLE DB provider and the SQL ODBC driver into one native dynamic-link library (DLL). It also provides new functionality above and beyond that supplied by the Windows Data Access Components (Windows DAC, formerly Microsoft Data Access Components, or MDAC). SQL Server Native Client can be used to create new applications or enhance existing applications that need to take advantage of features introduced in SQL Server 2005, such as multiple active result sets (MARS), user-defined data types (UDT), query notifications, snapshot isolation, and XML data type support.

> **NOTE**
>
> For a list of the differences between SQL Server Native Client and Windows DAC, plus information about issues to consider before updating a Windows DAC application to SQL Server Native Client, see Updating an Application to SQL Server Native Client from MDAC.

The SQL Server Native Client ODBC driver is always used in conjunction with the ODBC Driver Manager supplied with Windows DAC. The SQL Server Native Client OLE DB provider can be used in conjunction with OLE DB Core Services supplied with Windows DAC, but this is not a requirement; the choice to use Core Services or not depends on the requirements of the individual application (for example, if connection pooling is required).

ActiveX Data Object (ADO) applications may use the SQL Server Native Client OLE DB provider, but it is recommended to use ADO in conjunction with the **DataTypeCompatibility** connection string keyword (or its corresponding **DataSource** property). When using the SQL Server Native Client OLE DB provider, ADO applications may exploit those new features introduced in SQL Server 2005 that are available via the SQL Server Native Client via connection string keywords or OLE DB properties or Transact-SQL. For more information about the use of these features with ADO, see Using ADO with SQL Server Native Client.

SQL Server Native Client was designed to provide a simplified method of gaining native data access to SQL Server using either OLE DB or ODBC. It is simplified in that it combines OLE DB and ODBC technologies into one library, and it provides a way to innovate and evolve new data access features without changing the current Windows DAC components, which are now part of the Microsoft Windows platform.

While SQL Server Native Client uses components in Windows DAC, it is not explicitly dependant on a particular version of Windows DAC. You can use SQL Server Native Client with the version of Windows DAC that is installed with any operating system supported by SQL Server Native Client.

## In This Section

SQL Server Native Client
Lists the significant new SQL Server Native Client features.

When to Use SQL Server Native Client
Discusses how SQL Server Native Client fits in with Microsoft data access technologies, how it compares to Windows DAC and ADO.NET, and provides pointers for deciding which data access technology to use.

SQL Server Native Client Features

Describes the features supported by SQL Server Native Client.

Building Applications with SQL Server Native Client
Provides an overview of SQL Server Native Client development, including how it differs from Windows DAC, the components that it uses, and how ADO can be used with it.

This section also discusses SQL Server Native Client installation and deployment, including how to redistribute the SQL Server Native Client library.

System Requirements for SQL Server Native Client
Discusses the system resources needed to use SQL Server Native Client.

SQL Server Native Client (OLE DB)
Provides information about using the SQL Server Native Client OLE DB provider.

SQL Server Native Client (ODBC)
Provides information about using the SQL Server Native Client ODBC driver.

Finding More SQL Server Native Client Information
Provides additional resources about SQL Server Native Client, including links to external resources and getting further assistance.

SQL Server Native Client Errors
Contains topics about runtime errors associated with SQL Server Native Client.

## See Also

Updating an Application from SQL Server 2005 Native Client
ODBC How-to Topics
OLE DB How-to Topics

# SQL Server Management Objects (SMO) Programming Guide

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

SQL Server Management Objects (SMO) is a collection of objects that are designed for programming all aspects of managing Microsoft SQL Server. SQL Server Replication Management Objects (RMO) is a collection of objects that encapsulates SQL Server replication management.

| TOPIC | DESCRIPTION |
| --- | --- |
| Getting Started in SMO | Provides information on how to start developing a SMO application |
| Creating SMO Programs<br><br>Programming Specific Tasks | Provides information about programming the SMO objects in the Microsoft.SqlServer.management, Microsoft.SqlServer.Management.NotificationServices, Microsoft.SqlServer.Management.Smo, Microsoft.SqlServer.Management.Smo.Agent, Microsoft.SqlServer.Management.Smo.Broker, Microsoft.SqlServer.Management.Smo.Mail, Microsoft.SqlServer.Management.Smo.RegisteredServers, Microsoft.SqlServer.Management.Smo.Wmi, and Microsoft.SqlServer.Management.Trace namespaces.<br><br>This includes instructions to write programs that define databases and manage SQL Server. You can use SMO to create databases, perform backups, create jobs, configure SQL Server, assign permissions, and to perform many other administrative tasks. |
| Replication Developer Documentation | Provides information about programming the RMO objects in the Microsoft.SqlServer.Replication namespace. |

## See Also

Replication Developer Documentation

# Requirements for Running SQLXML Examples

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

The SQLXML 4.0 Programming section provides numerous examples. This topic lists requirements for creating working samples from these examples.

To create working samples using the SQLXML 4.0 examples, you need to have the following installed.

- The AdventureWorks sample database. For more information, see AdventureWorks Sample Databases.
- Microsoft SQL Server Native Client. For more information, see Installing SQL Server Native Client.
- MDAC 2.6 or later

  In many examples, templates are used to specify XPath queries against the mapping XSD schema. For more information, see Using Annotated XSD Schemas in Queries (SQLXML 4.0).

# WMI Provider for Configuration Management

5/3/2018 • 2 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2008) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

The WMI provider is a published layer that is used with the SQL Server Configuration Manager snap-in for Microsoft Management Console (MMC) and the Microsoft SQL Server Configuration Manager. It provides a unified way for interfacing with the API calls that manage the registry operations requested by SQL Server Configuration Manager and provides enhanced control and manipulation over the selected SQL Server services.

The SQL Server WMI Provider is a DLL and a MOF file, which are compiled automatically by SQL Server Setup.

The SQL Server WMI Provider contains a set of object classes used to control the SQL Server services using the following methods:

- A script language such as VBScript, JScript, or Perl, in which Windows Query Language (WQL) can be embedded.
- The ManagedComputer object in an SMO managed code program.
- The SQL Server Configuration Manager or MMC with the SQL Server WMI provider snap-in.

## Using a Script Language

The advantages of using a script language include the following:

- A development environment is not required.
- The files that support the script language are widely available.

  The script can also work with other WMI Providers in addition to the SQL Server WMI Provider. A domain administrator can use a script to set up the services, network settings, and alias settings on multiple computers on a network.

  This section deals with accessing the WMI Provider for Configuration Management from scripts in further detail.

## Using the SMO ManagedComputer Object

The ManagedComputer object is a managed SMO object that provides access to the WMI Provider for Configuration Management. By using an SMO program, the ManagedComputer object can be used to view and modify SQL Server services, network settings, and alias settings. For more information, see Managing Services and Network Settings by Using WMI Provider.

## Using the Microsoft Management Console or SQL Server Configuration Manager

Microsoft Management Console (MMC) provides an interface to manage SQL Server services, as opposed to a scripting language or managed code program. The SQL Server Management MMC snap-in can be used to stop and start services, and to change service accounts.

The SQL Server Configuration Manager can also be used to manage SQL Server services, client and server protocols, and server aliases

## See Also

Understanding the WMI Provider for Configuration Management
Working with the WMI Provider for Configuration Management
Using WQL and Scripting Languages with the WMI Provider for Configuration Management

# WMI Provider for Configuration Management Classes

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2008) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

The Windows Management Instrumentation (WMI) provider for Configuration Management provides functionality for the following classes:

CInstance Class

ClientNetLibInfo Class

ClientNetworkProtocol Class

ClientNetworkProtocolProperty Class

ClientSettings Class

ClientSettingsGeneralFlag Class

SecurityCertificate Class

ServerNetworkProtocol Class

ServerNetworkProtocolIPAddress Class

ServerNetworkProtocolProperty Class

ServerSettings Class

ServerSettingsGeneralFlag Class

SInstance Class

SqlErrorLogEvent Class

SqlErrorLogFile Class

SqlServerAlias Class

SqlService Class

SqlServiceAdvancedProperty Class

# WMI Provider for Server Events Concepts

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2008) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

The WMI Provider for Server Events lets you use Windows Management Instrumentation (WMI) to monitor events in an instance of SQL Server.

## In This Section

Understanding the WMI Provider for Server Events
Provides an overview of the provider architecture and how SQL Server Agent can be programmed to work with it.

Working with the WMI Provider for Server Events
Explains the factors that you need to consider before programming with the provider.

Using WQL with the WMI Provider for Server Events
Explains the WMI Query Language (WQL) syntax and how to use it when you program against the provider.

Sample: Creating a SQL Server Agent Alert by Using the WMI Provider for Server Events
Provides an example of using the WMI Provider to return trace event information on which to create a SQL Server Agent alert.

Sample: Use the WMI Event Provider with the .NET Framework
Provides an example of using the WMI Provider to return event data in a C# application.

WMI Provider for Server Events Classes and Properties
Introduces the event classes and properties that make up the programming mode of the provider.

# Errors and Events Reference (Database Engine)

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

This section contains selected Database Engine error messages that need further explanation.

## In This Section

Database Engine Events and Errors

Describes the format of Database Engine error messages and explains how to view error messages and return error messages to applications.

Provides an explanation of Database Engine error messages, possible causes, and any actions you can take to correct the problem.

## External Resources

If you have not found the information you are looking for in the product documentation or on the Web, you can either ask a question in the SQL Server community or request help from Microsoft support.

The following table links to and describes these resources.

| RESOURCE | DESCRIPTION |
| --- | --- |
| SQL Server Community | This site has links to newsgroups and forums monitored by the SQL Server community. It also lists community information sources, such as blogs and Web sites. The SQL Server community is very helpful in answering questions, although an answer cannot be guaranteed. |
| SQL Server Developer Center Community | This site focuses on the newsgroups, forums, and other community resources that are useful to SQL Server developers. |
| Microsoft Help and Support | You can use this Web site to open a case with a Microsoft support professional. |

# SQL Server Event Class Reference

5/3/2018 • 5 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

SQL Server Profiler lets you record events as they occur in an instance of the Microsoft SQL Server Database Engine. The recorded events are instances of the event classes in the trace definition. In SQL Server Profiler, event classes and their event categories are available on the **Events Selection** tab of the **Trace File Properties** dialog box.

The following table describes the event categories and lists their associated event classes.

| EVENT CATEGORY | EVENT CLASSES |
|---|---|
| The Broker Event Category includes event classes that are produced by the Service Broker. | Broker:Activation Event Class<br>Broker:Connection Event Class<br>Broker:Conversation Event Class<br>Broker:Conversation Group Event Class<br>Broker:Corrupted Message Event Class<br>Broker:Forwarded Message Dropped Event Class<br>Broker:Forwarded Message Sent Event Class<br>Broker:Message Classify Event Class<br>Broker:Message Drop Event Class<br>Broker:Remote Message Ack Event Class |
| The Cursors Event Category includes event classes that are produced by cursor operations. | CursorClose Event Class<br>CursorExecute Event Class<br>CursorImplicitConversion Event Class<br>CursorOpen Event Class<br>CursorPrepare Event Class<br>CursorRecompile Event Class<br>CursorUnprepare Event Class |
| The CLR Event Category includes event classes that are produced by the execution of .NET common language runtime (CLR) objects. | Assembly Load Event Class |

| EVENT CATEGORY | EVENT CLASSES |
| --- | --- |
| The Database Event Category includes event classes that are produced when data or log files grow or shrink automatically. | Data File Auto Grow Event Class<br><br>Data File Auto Shrink Event Class<br><br>Database Mirroring State Change Event Class<br><br>Log File Auto Grow Event Class<br><br>Log File Auto Shrink Event Class |
| The Deprecation Event Category includes deprecation related events. | Deprecation Announcement Event Class<br><br>Deprecation Final Support Event Class |
| The Errors and Warnings Event Category (Database Engine) includes event classes that are produced when a SQL Server error or warning is returned, for example, if an error occurs during the compilation of a stored procedure or an exception occurs in SQL Server. | Attention Event Class<br><br>Background Job Error Event Class<br><br>Blocked Process Report Event Class<br><br>CPU Threshold Exceeded Event Class<br><br>ErrorLog Event Class<br><br>EventLog Event Class<br><br>Exception Event Class<br><br>Exchange Spill Event Class<br><br>Execution Warnings Event Class<br><br>Hash Warning Event Class<br><br>Missing Column Statistics Event Class<br><br>Missing Join Predicate Event Class<br><br>Sort Warnings Event Class<br><br>User Error Message Event Class |
| The Full Text Event Category includes event classes that are produced when full-text searches are started, interrupted, or stopped. | FT:Crawl Aborted Event Class<br><br>FT:Crawl Started Event Class<br><br>FT:Crawl Stopped Event Class |

| EVENT CATEGORY | EVENT CLASSES |
| --- | --- |
| The Locks Event Category includes event classes that are produced when a lock is acquired, cancelled, released, or has some other action performed on it. | Deadlock Graph Event Class<br><br>Lock:Acquired Event Class<br><br>Lock:Cancel Event Class<br><br>Lock:Deadlock Chain Event Class<br><br>Lock:Deadlock Event Class<br><br>Lock:Escalation Event Class<br><br>Lock:Released Event Class<br><br>Lock:Timeout (timeout > 0) Event Class<br><br>Lock:Timeout Event Class |
| The Objects Event Category includes event classes that are produced when database objects are created, opened, closed, dropped, or deleted. | Auto Stats Event Class<br><br>Object:Altered Event Class<br><br>Object:Created Event Class<br><br>Object:Deleted Event Class |
| The OLEDB Event Category includes event classes that are produced by OLE DB calls. | OLEDB Call Event Class<br><br>OLEDB DataRead Event Class<br><br>OLEDB Errors Event Class<br><br>OLEDB Provider Information Event Class<br><br>OLEDB QueryInterface Event Class |
| The Performance Event Category includes event classes that are produced when SQL data manipulation language (DML) operators execute. | Degree of Parallelism (7.0 Insert) Event Class<br><br>Performance Statistics Event Class<br><br>Showplan All Event Class<br><br>Showplan All for Query Compile Event Class<br><br>Showplan Statistics Profile Event Class<br><br>Showplan Text Event Class<br><br>Showplan Text (Unencoded) Event Class<br><br>Showplan XML Event Class<br><br>Showplan XML for Query Compile Event Class<br><br>Showplan XML Statistics Profile Event Class<br><br>SQL:FullTextQuery Event Class |

| EVENT CATEGORY | EVENT CLASSES |
| --- | --- |
| The Progress Report Event Category includes the **Progress Report: Online Index Operation** event class. | Progress Report: Online Index Operation Event Class |
| The Scans Event Category includes event classes that are produced when tables and indexes are scanned. | Scan:Started Event Class<br><br>Scan:Stopped Event Class |
| The Security Audit Event Category includes event classes that are used to audit server activity. | Audit Add DB User Event Class<br><br>Audit Add Login to Server Role Event Class<br><br>Audit Add Member to DB Role Event Class<br><br>Audit Add Role Event Class<br><br>Audit Addlogin Event Class<br><br>Audit App Role Change Password Event Class<br><br>Audit Backup and Restore Event Class<br><br>Audit Broker Conversation Event Class<br><br>Audit Broker Login Event Class<br><br>Audit Change Audit Event Class<br><br>Audit Change Database Owner Event Class<br><br>Audit Database Management Event Class<br><br>Audit Database Object Access Event Class<br><br>Audit Database Object GDR Event Class<br><br>Audit Database Object Management Event Class<br><br>Audit Database Object Take Ownership Event Class<br><br>Audit Database Operation Event Class<br><br>Audit Database Principal Impersonation Event Class<br><br>Audit Database Principal Management Event Class<br><br>Audit Database Scope GDR Event Class<br><br>Audit DBCC Event Class<br><br>Audit Login Change Password Event Class<br><br>Audit Login Change Property Event Class<br><br>Audit Login Event Class<br><br>Audit Login Failed Event Class<br><br>Audit Login GDR Event Class<br><br>Audit Logout Event Class<br><br>Audit Object Derived Permission Event Class |

| EVENT CATEGORY | EVENT CLASSES |
|---|---|
| | Audit Object Derived Permission Event Class<br>Audit Schema Object Access Event Class<br><br>Audit Schema Object GDR Event Class<br><br>Audit Schema Object Management Event Class<br><br>Audit Schema Object Take Ownership Event Class<br><br>Audit Server Alter Trace Event Class<br><br>Audit Server Object GDR Event Class<br><br>Audit Server Object Management Event Class<br><br>Audit Server Object Take Ownership Event Class<br><br>Audit Server Operation Event Class<br><br>Audit Server Principal Impersonation Event Class<br><br>Audit Server Principal Management Event Class<br><br>Audit Server Scope GDR Event Class<br><br>Audit Server Starts and Stops Event Class<br><br>Audit Statement Permission Event Class |
| The Server Event Category contains general server events. | Mount Tape Event Class<br><br>Server Memory Change Event Class<br><br>Trace File Close Event Class |
| The Sessions Event Category includes event classes produced by clients connecting to and disconnecting from an instance of SQL Server. | ExistingConnection Event Class |

| EVENT CATEGORY | EVENT CLASSES |
| --- | --- |
| The Stored Procedures Event Category includes event classes produced by the execution of stored procedures. | PreConnect:Completed Event Class<br><br>PreConnect:Starting Event Class<br><br>RPC:Completed Event Class<br><br>RPC Output Parameter Event Class<br><br>RPC:Starting Event Class<br><br>SP:CacheHit Event Class<br><br>SP:CacheInsert Event Class<br><br>SP:CacheMiss Event Class<br><br>SP:CacheRemove Event Class<br><br>SP:Completed Event Class<br><br>SP:Recompile Event Class<br><br>SP:Starting Event Class<br><br>SP:StmtCompleted Event Class<br><br>SP:StmtStarting Event Class |
| The Transactions Event Category includes event classes produced by the execution of Microsoft Distributed Transaction Coordinator transactions or by writing to the transaction log. | DTCTransaction Event Class<br><br>SQLTransaction Event Class<br><br>TM: Begin Tran Completed Event Class<br><br>TM: Begin Tran Starting Event Class<br><br>TM: Commit Tran Completed Event Class<br><br>TM: Commit Tran Starting Event Class<br><br>TM: Promote Tran Completed Event Class<br><br>TM: Promote Tran Starting Event Class<br><br>TM: Rollback Tran Completed Event Class<br><br>TM: Rollback Tran Starting Event Class<br><br>TM: Save Tran Completed Event Class<br><br>TM: Save Tran Starting Event Class<br><br>TransactionLog Event Class |

| EVENT CATEGORY | EVENT CLASSES |
| --- | --- |
| The TSQL Event Category includes event classes produced by the execution of Transact-SQL statements passed to an instance of SQL Server from the client. | Exec Prepared SQL Event Class<br><br>Prepare SQL Event Class<br><br>SQL:BatchCompleted Event Class<br><br>SQL:BatchStarting Event Class<br><br>SQL:StmtCompleted Event Class<br><br>SQL:StmtRecompile Event Class<br><br>SQL:StmtStarting Event Class<br><br>Unprepare SQL Event Class<br><br>XQuery Static Type Event Class |
| The User-Configurable Event Category includes event classes that you can define. | User-Configurable Event Class |

## See Also

SQL Server Profiler

# Extended Events

5/3/2018 • 5 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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.

You can find more information about Extended Events at:

- Quick Start: Extended events in SQL Server
- Blogs: SQL Server Extended Events

## 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.

## Extended Events Concepts

SQL Server Extended Events (Extended Events) builds on existing concepts, such as an event or an event consumer, uses concepts from Event Tracing for Windows, and introduces new concepts.

The following table describes the concepts in Extended Events.

| TOPIC | DESCRIPTION |
| --- | --- |
| SQL Server Extended Events Packages | Describes the Extended Events packages that contain objects that are used for obtaining and processing data when an Extended Events session is running. |
| SQL Server Extended Events Targets | Describes the event consumers that can receive data during an event session. |
| SQL Server Extended Events Engine | Describes the engine that implements and manages an Extended Events session. |
| SQL Server Extended Events Sessions | Describes the Extended Events session. |

## Extended Events Architecture

Extended Events (Extended Events) is a general event-handling system for server systems. The Extended Events infrastructure supports the correlation of data from SQL Server, and under certain conditions, the correlation of data from the operating system and database applications. In the latter case, Extended Events output must be directed to Event Tracing for Windows (ETW) to correlate the event data with operating system or application event data.

All applications have execution points that are useful both inside and outside an application. Inside the application, asynchronous processing may be enqueued using information that is collected during the initial execution of a task. Outside the application, execution points provide monitoring utilities with information about the behavioral

and performance characteristics of the monitored application.

Extended Events supports using event data outside a process. This data is typically used by:

- Tracing tools, such as SQL Trace and System Monitor.
- Logging tools, such as the Windows event log or the SQL Server error log.
- Users administering a product or developing applications on a product.

  Extended Events has the following key design aspects:

- The Extended Events engine is event agnostic. This enables the engine to bind any event to any target because the engine is not constrained by event content. For more information about the Extended Events engine, see SQL Server Extended Events Engine.
- Events are separated from event consumers, which are called *targets* in Extended Events. This means that any target can receive any event. In addition, any event that is raised can be automatically consumed by the target, which can log or provide additional event context. For more information, see SQL Server Extended Events Targets.
- Events are distinct from the action to take when an event occurs. Therefore, any action can be associated with any event.
- Predicates can dynamically filter when event data should be captured. This adds to the flexibility of the Extended Events infrastructure. For more information, see SQL Server Extended Events Packages.

  Extended Events can synchronously generate event data (and asynchronously process that data) which provides a flexible solution for event handling. In addition, Extended Events provides the following features:

- A unified approach to handling events across the server system, while enabling users to isolate specific events for troubleshooting purposes.
- Integration with, and support for existing ETW tools.
- A fully configurable event handling mechanism that is based on Transact-SQL.
- The ability to dynamically monitor active processes, while having minimal effect on those processes.
- A default system health session that runs without any noticeable performance effects. The session collects system data that you can use to help troubleshoot performance issues. For more information, see Use the system_health Session.

## Extended Events Tasks

Using Management Studio or Transact-SQL to execute Transact-SQL Data Definition Language (DDL) statements, dynamic management views and functions, or catalog views, you can create simple or complex SQL Server Extended Events troubleshooting solutions for your SQL Server environment.

| TASK DESCRIPTION | TOPIC |
| --- | --- |
| Use the **Object Explorer** to manage event sessions. | Manage Event Sessions in the Object Explorer |
| Describes how to create an Extended Events session. | Create an Extended Events Session |
| Describes how to view and refresh target data. | Advanced Viewing of Target Data from Extended Events in SQL Server |

| TASK DESCRIPTION | TOPIC |
| --- | --- |
| Describes how to use Extended Events tools to create and manage your SQL Server Extended Events sessions. | Extended Events Tools |
| Describes how to alter an Extended Events session. | Alter an Extended Events Session |
| Describes how to get information about the fields associated with the events. | Get the Fields for All Events |
| Describes how to find out what events are available in the registered packages. | View the Events for Registered Packages |
| Describes how to determine what Extended Events targets are available in the registered packages. | View the Extended Events Targets for Registered Packages |
| Describes how to view the Extended Events events and actions that are equivalent to each SQL Trace event and its associated columns. | View the Extended Events Equivalents to SQL Trace Event Classes |
| Describes how to find the parameters you can set when you use the ADD TARGET argument in CREATE EVENT SESSION or ALTER EVENT SESSION. | Get the Configurable Parameters for the ADD TARGET Argument |
| Describes how to convert an existing SQL Trace script to an Extended Events session. | Convert an Existing SQL Trace Script to an Extended Events Session |
| Describes how to determine which queries are holding the lock, the plan of the query, and the Transact-SQL stack at the time the lock was taken. | Determine Which Queries Are Holding Locks |
| Describes how to identify the source of locks that are hindering database performance. | Find the Objects That Have the Most Locks Taken on Them |
| Describes how to use Extended Events with Event Tracing for Windows to monitor system activity. | Monitor System Activity Using Extended Events |
| Using the Catalog views and the Dynamic management views (DMVs) for extended events | SELECTs and JOINs From System Views for Extended Events in SQL Server |

## See Also

Data-tier Applications
DAC Support For SQL Server Objects and Versions
Deploy a Data-tier Application
Monitor Data-tier Applications
Extended Events Dynamic Management Views
Extended Events Catalog Views (Transact-SQL)

# Graph processing with SQL Server and Azure SQL Database

5/3/2018 • 2 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2017) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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.

## Graph features introduced in SQL Server 2017 (14.x)

We are starting to add graph extensions to SQL Server, to make storing and querying graph data easier. Following features are introduced in the first release.

### Create graph objects

Transact-SQL extensions will allow users to create node or edge tables. Both nodes and edges can have properties associated to them. Since, nodes and edges are stored as tables, all the operations that are supported on relational tables are supported on node or edge table. Here is an example:

```
CREATE TABLE Person (ID INTEGER PRIMARY KEY, Name VARCHAR(100), Age INT) AS NODE;
CREATE TABLE friends (StartDate date) AS EDGE;
```

Node Properties

| $node_id | Name | Age |
|---|---|---|
| {"type":"node","id":0} | John | 30 |
| "type":"node","id":1} | Mary | 28 |
| "type":"node","id":2} | Alice | 25 |

**Person Node Table**

Nodes that this edge connects; Edge Properties

| $edge_id | $from_id | $to_id | StartDate |
|---|---|---|---|
| {"type":"edge", "id":0} | {"type":"node","id":0} | {"type":"node","id":1} | 01/01/2013 |
| {"type":"edge", "id":1} | {"type":"node","id":1} | {"type":"node","id":2} | 05/05/2010 |
| {"type":"edge", "id":2} | {"type":"node","id":2} | {"type":"node","id":0} | 09/09/2016 |

**Friends Edge Table**

Nodes and Edges are stored as tables

### Query language extensions

New `MATCH` clause is introduced to support pattern matching and multi-hop navigation through the graph. The `MATCH` function uses ASCII-art style syntax for pattern matching. For example:

```
-- Find friends of John
SELECT Person2.Name
FROM Person Person1, Friends, Person Person2
WHERE MATCH(Person1-(Friends)->Person2)
AND Person1.Name = 'John';
```

### Fully integrated in SQL Server Engine

Graph extensions are fully integrated in SQL Server engine. We use the same storage engine, metadata, query processor, etc. to store and query graph data. This enables users to query across their graph and relational data in a single query. Users can also benefit from combining graph capabilities with other SQL Server technologies like columnstore, HA, R services, etc. SQL graph database also supports all the security and compliance features available with SQL Server.

### Tooling and ecosystem

Users benefit from existing tools and ecosystem that SQL Server offers. Tools like backup and restore, import and export, BCP just work out of the box. Other tools or services like SSIS, SSRS or PowerBI will work with graph tables, just the way they work with relational tables.

## Next steps

Read the SQL Graph Database - Architecture

# Hierarchical Data (SQL Server)

5/3/2018 • 13 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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. Examples of the hierarchical data that is commonly stored in databases include the following:

- An organizational structure
- A file system
- A set of tasks in a project
- A taxonomy of language terms
- A graph of links between Web pages

  Use hierarchyid as a data type to create tables with a hierarchical structure, or to describe the hierarchical structure of data that is stored in another location. Use the hierarchyid functions in Transact-SQL to query and manage hierarchical data.

## Key Properties of hierarchyid

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

- 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 slightly less compact representation.

## 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:

```
USE AdventureWorks2012 ;
GO

CREATE TABLE ParentChildOrg
   (
    BusinessEntityID int PRIMARY KEY,
    ManagerId int REFERENCES ParentChildOrg(BusinessEntityID),
    EmployeeName nvarchar(50)
   ) ;
GO
```

Comparing Parent/Child and **hierarchyid** for Common Operations

- Subtree queries are significantly faster with **hierarchyid**.
- Direct descendant queries are slightly slower with **hierarchyid**.
- Moving non-leaf nodes is slower with **hierarchyid**.
- Inserting non-leaf nodes and inserting or moving leaf nodes has the same complexity with **hierarchyid**.

  Parent/Child might be superior when the following conditions exist:

- The size of the key is critical. For the same number of nodes, a **hierarchyid** value is equal to or larger than an integer-family (**smallint**, **int**, **bigint**) value. This is only a reason to use Parent/Child in rare cases, because **hierarchyid** has significantly better locality of I/O and CPU complexity than the common table expressions required when you are using a Parent/Child structure.
- Queries rarely query across sections of the hierarchy. In other words, queries usually address only a single

point in the hierarchy. In these cases co-location is not important. For example, Parent/Child is superior when the organization table is only used to process payroll for individual employees.

- Non-leaf subtrees move frequently and performance is very important. In a parent/child representation changing the location of a row in a hierarchy affects a single row. Changing the location of a row in a **hierarchyid** usage affects *n* rows, where *n* is number of nodes in the sub-tree being moved.

  If the non-leaf subtrees move frequently and performance is important, but most of the moves are at a well-defined level of the hierarchy, consider splitting the higher and lower levels into two hierarchies. This makes all moves into leaf-levels of the higher hierarchy. For instance, consider a hierarchy of Web sites hosted by a service. Sites contain many pages arranged in a hierarchical manner. Hosted sites might be moved to other locations in the site hierarchy, but the subordinate pages are rarely re-arranged. This could be represented via:

  ```
  CREATE TABLE HostedSites
     (
      SiteId hierarchyid, PageId hierarchyid
     ) ;
  GO
  ```

**XML**

An XML document is a tree, and therefore a single XML data type instance can represent a complete hierarchy. In SQL Server when an XML index is created, **hierarchyid** values are used internally to represent the position in the hierarchy.

Using XML data type can be superior when all the following are true:

- The complete hierarchy is always stored and retrieved.
- The data is consumed in XML format by the application.
- Predicate searches are extremely limited and not performance critical.

  For example, if an application tracks multiple organizations, always stores and retrieves the complete organizational hierarchy, and does not query into a single organization, a table of the following form might make sense:

```
CREATE TABLE XMLOrg
    (
    Orgid int,
    Orgdata xml
    ) ;
GO
```

# 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. For a detailed example of indexing strategies, see Tutorial: Using the hierarchyid Data Type.

**Creating Indexes**

The GetLevel() method can be used to create a breadth first ordering. In the following example, both breadth-first and depth-first indexes are created:

```
USE AdventureWorks2012 ;
GO

CREATE TABLE Organization
   (
    BusinessEntityID hierarchyid,
    OrgLevel as BusinessEntityID.GetLevel(),
    EmployeeName nvarchar(50) NOT NULL
   ) ;
GO

CREATE CLUSTERED INDEX Org_Breadth_First
ON Organization(OrgLevel,BusinessEntityID) ;
GO

CREATE UNIQUE INDEX Org_Depth_First
ON Organization(BusinessEntityID) ;
GO
```

# Examples

**Simple Example**

The following example is intentionally simplistic to help you get started. First create a table to hold some geography data.

```
CREATE TABLE SimpleDemo
(Level hierarchyid NOT NULL,
Location nvarchar(30) NOT NULL,
LocationType nvarchar(9) NULL);
```

Now insert data for some continents, countries, states, and cities.

```
INSERT SimpleDemo
VALUES
('/1/', 'Europe', 'Continent'),
('/2/', 'South America', 'Continent'),
('/1/1/', 'France', 'Country'),
('/1/1/1/', 'Paris', 'City'),
('/1/2/1/', 'Madrid', 'City'),
('/1/2/', 'Spain', 'Country'),
('/3/', 'Antarctica', 'Continent'),
('/2/1/', 'Brazil', 'Country'),
('/2/1/1/', 'Brasilia', 'City'),
('/2/1/2/', 'Bahia', 'State'),
('/2/1/2/1/', 'Salvador', 'City'),
('/3/1/', 'McMurdo Station', 'City');
```

Select the data, adding a column that converts the Level data into a text value that is easy to understand. This query also orders the result by the **hierarchyid** data type.

```
SELECT CAST(Level AS nvarchar(100)) AS [Converted Level], *
FROM SimpleDemo ORDER BY Level;
```

Here is the result set.

```
Converted Level  Level     Location         LocationType
/1/              0x58      Europe           Continent
/1/1/            0x5AC0    France           Country
/1/1/1/          0x5AD6    Paris            City
/1/2/            0x5B40    Spain            Country
/1/2/1/          0x5B56    Madrid           City
/2/              0x68      South America    Continent
/2/1/            0x6AC0    Brazil           Country
/2/1/1/          0x6AD6    Brasilia         City
/2/1/2/          0x6ADA    Bahia            State
/2/1/2/1/        0x6ADAB0  Salvador         City
/3/              0x78      Antarctica       Continent
/3/1/            0x7AC0    McMurdo Station  City
```

Notice that the hierarchy has a valid structure, even though it is not internally consistent. Bahia is the only state. It appears in the hierarchy as a peer of the city Brasilia. Similarly, McMurdo Station does not have a parent country. Users must decide if this type of hierarchy is appropriate for their use.

Add another row and select the results.

```
INSERT SimpleDemo
VALUES ('/1/3/1/', 'Kyoto', 'City'), ('/1/3/1/', 'London', 'City');
SELECT CAST(Level AS nvarchar(100)) AS [Converted Level], * FROM SimpleDemo ORDER BY Level;
```

This demonstrates more possible problems. Kyoto can be inserted as level `/1/3/1/` even though there is no parent level `/1/3/`. And both London and Kyoto have the same value for the **hierarchyid**. Again, users must decide if this type of hierarchy is appropriate for their use, and block values that are invalid for their usage.

Also, this table did not use the top of the hierarchy `'/'`. It was omitted because there is no common parent of all the continents. You can add one by adding the whole planet.

```
INSERT SimpleDemo
VALUES ('/', 'Earth', 'Planet');
```

## Related Tasks

### Migrating from Parent/Child to hierarchyid

Most trees are represented using Parent/Child. The easiest way to migrate from a Parent/Child structure to a table using **hierarchyid** is to use a temporary column or a temporary table to keep track of the number of nodes at each level of the hierarchy. For an example of migrating a Parent/Child table, see lesson 1 of Tutorial: Using the hierarchyid Data Type.

### Managing a Tree Using hierarchyid

Although a **hierarchyid** column does not necessarily represent a tree, an application can easily ensure that it does.

- When generating new values, do one of the following:
  - Keep track of the last child number in the parent row.
  - Compute the last child. Doing this efficiently requires a breadth-first index.
- Enforce uniqueness by creating a unique index on the column, perhaps as part of a clustering key. To ensure that unique values are inserted, do one of the following:
  - Detect unique key violation failures and retry.
  - Determine the uniqueness of each new child node, and insert it as part of a serializable transaction.

#### Example Using Error Detection

In the following example, the sample code computes the new child **EmployeeId** value, and then detects any key violation and returns to **INS_EMP** marker to recompute the **EmployeeId** value for the new row:

```
USE AdventureWorks ;
GO

CREATE TABLE Org_T1
   (
    EmployeeId hierarchyid PRIMARY KEY,
    OrgLevel AS EmployeeId.GetLevel(),
    EmployeeName nvarchar(50)
   ) ;
GO

CREATE INDEX Org_BreadthFirst ON Org_T1(OrgLevel, EmployeeId)
GO

CREATE PROCEDURE AddEmp(@mgrid hierarchyid, @EmpName nvarchar(50) )
AS
BEGIN
    DECLARE @last_child hierarchyid
INS_EMP:
    SELECT @last_child = MAX(EmployeeId) FROM Org_T1
    WHERE EmployeeId.GetAncestor(1) = @mgrid
INSERT Org_T1 (EmployeeId, EmployeeName)
SELECT @mgrid.GetDescendant(@last_child, NULL), @EmpName
-- On error, return to INS_EMP to recompute @last_child
IF @@error <> 0 GOTO INS_EMP
END ;
GO
```

#### Example Using a Serializable Transaction

The **Org_BreadthFirst** index ensures that determining **@last_child** uses a range seek. In addition to other error cases an application might want to check, a duplicate key violation after the insert indicates an attempt to add multiple employees with the same id, and therefore **@last_child** must be recomputed. The following code uses a serializable transaction and a breadth-first index to compute the new node value:

```
CREATE TABLE Org_T2
    (
    EmployeeId hierarchyid PRIMARY KEY,
    LastChild hierarchyid,
    EmployeeName nvarchar(50)
    ) ;
GO

CREATE PROCEDURE AddEmp(@mgrid hierarchyid, @EmpName nvarchar(50))
AS
BEGIN
DECLARE @last_child hierarchyid
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRANSACTION

UPDATE Org_T2
SET @last_child = LastChild = EmployeeId.GetDescendant(LastChild,NULL)
WHERE EmployeeId = @mgrid
INSERT Org_T2 (EmployeeId, EmployeeName)
    VALUES(@last_child, @EmpName)
COMMIT
END ;
```

The following code populates the table with three rows and returns the results:

```
INSERT Org_T2 (EmployeeId, EmployeeName)
    VALUES(hierarchyid::GetRoot(), 'David') ;
GO
AddEmp 0x , 'Sariya'
GO
AddEmp 0x58 , 'Mary'
GO
SELECT * FROM Org_T2
```

Here is the result set.

```
EmployeeId LastChild EmployeeName
---------- --------- ------------
0x         0x58      David
0x58       0x5AC0    Sariya
0x5AC0     NULL      Mary
```

**Enforcing a tree**

The above examples illustrate how an application can ensure that a tree is maintained. To enforce a tree by using constraints, a computed column that defines the parent of each node can be created with a foreign key constraint back to the primary key id.

```
CREATE TABLE Org_T3
(
   EmployeeId hierarchyid PRIMARY KEY,
   ParentId AS EmployeeId.GetAncestor(1) PERSISTED
      REFERENCES Org_T3(EmployeeId),
   LastChild hierarchyid,
   EmployeeName nvarchar(50)
)
GO
```

This method of enforcing a relationship is preferred when code that is not trusted to maintain the hierarchical tree has direct DML access to the table. However this method might reduce performance because the constraint must

be checked on every DML operation.

**Finding Ancestors by Using the CLR**

A common operation involving two nodes in a hierarchy is to find the lowest common ancestor. This can be written in either Transact-SQL or CLR, because the **hierarchyid** type is available in both. CLR is recommended because performance will be faster.

Use the following CLR code to list ancestors and to find the lowest common ancestor:

```
using System;
using System.Collections;
using System.Text;
using Microsoft.SqlServer.Server;
using Microsoft.SqlServer.Types;

public partial class HierarchyId_Operations
{
    [SqlFunction(FillRowMethodName = "FillRow_ListAncestors")]
    public static IEnumerable ListAncestors(SqlHierarchyId h)
    {
        while (!h.IsNull)
        {
            yield return (h);
            h = h.GetAncestor(1);
        }
    }
    public static void FillRow_ListAncestors(Object obj, out SqlHierarchyId ancestor)
    {
        ancestor = (SqlHierarchyId)obj;
    }

    public static HierarchyId CommonAncestor(SqlHierarchyId h1, HierarchyId h2)
    {
        while (!h1.IsDescendant(h2))
            h1 = h1.GetAncestor(1);

        return h1;
    }
}
```

To use the **ListAncestor** and **CommonAncestor** methods in the following Transact-SQL examples, build the DLL and create the **HierarchyId_Operations** assembly in SQL Server by executing code similar to the following:

```
CREATE ASSEMBLY HierarchyId_Operations
FROM '<path to DLL>\ListAncestors.dll'
GO
```

**Listing Ancestors**

Creating a list of ancestors of a node is a common operation, for instance to show position in an organization. One way of doing this is by using a table-valued-function using the **HierarchyId_Operations** class defined above:

Using Transact-SQL:

```
CREATE FUNCTION ListAncestors (@node hierarchyid)
RETURNS TABLE (node hierarchyid)
AS
EXTERNAL NAME HierarchyId_Operations.HierarchyId_Operations.ListAncestors
GO
```

Example of usage:

```
DECLARE @h hierarchyid
SELECT @h = OrgNode
FROM HumanResources.EmployeeDemo
WHERE LoginID = 'adventure-works\janice0' -- /1/1/5/2/

SELECT LoginID, OrgNode.ToString() AS LogicalNode
FROM HumanResources.EmployeeDemo AS ED
JOIN ListAncestors(@h) AS A
   ON ED.OrgNode = A.Node
GO
```

#### Finding the Lowest Common Ancestor

Using the **HierarchyId_Operations** class defined above, create the following Transact-SQL function to find the lowest common ancestor involving two nodes in a hierarchy:

```
CREATE FUNCTION CommonAncestor (@node1 hierarchyid, @node2 hierarchyid)
RETURNS hierarchyid
AS
EXTERNAL NAME HierarchyId_Operations.HierarchyId_Operations.CommonAncestor
GO
```

Example of usage:

```
DECLARE @h1 hierarchyid, @h2 hierarchyid

SELECT @h1 = OrgNode
FROM  HumanResources.EmployeeDemo
WHERE LoginID = 'adventure-works\jossef0' -- Node is /1/1/3/

SELECT @h2 = OrgNode
FROM HumanResources.EmployeeDemo
WHERE LoginID = 'adventure-works\janice0' -- Node is /1/1/5/2/

SELECT OrgNode.ToString() AS LogicalNode, LoginID
FROM HumanResources.EmployeeDemo
WHERE OrgNode = dbo.CommonAncestor(@h1, @h2) ;
```

The resultant node is /1/1/

#### Moving Subtrees

Another common operation is moving subtrees. The procedure below takes the subtree of **@oldMgr** and makes it (including **@oldMgr**) a subtree of **@newMgr**.

```
CREATE PROCEDURE MoveOrg(@oldMgr nvarchar(256), @newMgr nvarchar(256) )
AS
BEGIN
DECLARE @nold hierarchyid, @nnew hierarchyid
SELECT @nold = OrgNode FROM HumanResources.EmployeeDemo WHERE LoginID = @oldMgr ;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRANSACTION
SELECT @nnew = OrgNode FROM HumanResources.EmployeeDemo WHERE LoginID = @newMgr ;

SELECT @nnew = @nnew.GetDescendant(max(OrgNode), NULL)
FROM HumanResources.EmployeeDemo WHERE OrgNode.GetAncestor(1)=@nnew ;

UPDATE HumanResources.EmployeeDemo
SET OrgNode = OrgNode.GetReparentedValue(@nold, @nnew)
WHERE OrgNode.IsDescendantOf(@nold) = 1 ;

COMMIT TRANSACTION
END ;
GO
```

# See Also

hierarchyid Data Type Method Reference
Tutorial: Using the hierarchyid Data Type
hierarchyid (Transact-SQL)

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

5/3/2018 • 2 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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. Typically you run these commands in SQL Server Management Studio (SSMS). For more info, see Import Bulk Data by Using BULK INSERT or OPENROWSET(BULK...).

### Use BCP from the command prompt

You can import and export data with the BCP command-line utility. For more info, see Import and Export Bulk Data by Using the bcp Utility.

### Use the Import Flat File Wizard

If you don't need all the configuration options available in the Import and Export Wizard and other tools, you can import a text file into SQL Server by using the **Import Flat File Wizard** in SQL Server Management Studio (SSMS). For more info, see the following articles:

- Import Flat File to SQL Wizard
- What's new in SQL Server Management Studio 17.3
- Introducing the new Import Flat File Wizard in SSMS 17.3

### Use the SQL Server Import and Export Wizard

You can import data to, or export data from, a variety of sources and destinations with the SQL Server Import and Export Wizard. To use the wizard, you must have SQL Server Integration Services (SSIS) or SQL Server Data Tools (SSDT) installed. For more info, see Import and Export Data with the SQL Server Import and Export Wizard.

### Design your own import or export

If you want to design a custom data import, you can use one of the following features or services:

- SQL Server Integration Services. For more info, see SQL Server Integration Services.
- Azure Data Factory. For more info, see Introduction to Azure Data Factory.

## Data formats for import and export

### Supported formats

You can import data from, and export data to, flat files or a variety of other file formats, relational databases, and cloud services. To learn more about these options for specific tools, see the following topics

- For the SQL Server Import and Export Wizard, see Connect to Data Sources with the SQL Server Import and

Export Wizard.

- For SQL Server Integration Services, see Integration Services (SSIS) Connections.
- For Azure Data Factory, see Azure Data Factory Connectors.

**Commonly used data formats**

There are special considerations and examples available for some commonly-used data formats. To learn more about these data formats, see the following topics:

- For Excel, see Import from Excel.
- For JSON, see Import JSON Documents.
- For XML, see Import and Export XML Documents.
- For Azure Blob Storage, see Import and Export from Azure Blob Storage.

# Next steps

If you're not sure where to begin with your import or export task, consider the SQL Server Import and Export Wizard. For a quick introduction, see Get started with this simple example of the Import and Export Wizard.

# In-Memory OLTP (In-Memory Optimization)

5/3/2018 • 2 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

In-Memory OLTP can significantly improve the performance of transaction processing, data ingestion and data load, and transient data scenarios. To jump into the basic code and knowledge you need to quickly test your own memory-optimized table and natively compiled stored procedure, see

- Quick Start 1: In-Memory OLTP Technologies for Faster Transact-SQL Performance.

A 17-minute video explaining In-Memory OLTP and demonstrating performance benefits:

- In-Memory OLTP in SQL Server 2016.

To download the performance demo for In-Memory OLTP used in the video:

- In-Memory OLTP Performance Demo v1.0

For a more detailed overview of In-Memory OLTP and a review of scenarios that see performance benefits from the technology:

- Overview and Usage Scenarios

  Note that In-Memory OLTP is the SQL Server technology for improving performance of transaction processing. For the SQL Server technology that improves reporting and analytical query performance see Columnstore Indexes Guide.

  Several improvements have been made to In-Memory OLTP in SQL Server 2016 (13.x) and SQL Server 2017 (14.x), as well as in Azure SQL Database. The Transact-SQL surface area has been increased to make it easier to migrate database applications. Support for performing ALTER operations for memory-optimized tables and natively compiled stored procedures has been added, to make it easier to maintain applications. For information about the new features in In-Memory OLTP, see Columnstore indexes - what's new.

> **NOTE**
>
> **Try it out**
>
> In-Memory OLTP is available in Premium and Business Critical tier Azure SQL databases and elastic pools. To get started with In-Memory OLTP, as well as Columnstore in Azure SQL Database, see Optimize Performance using In-Memory Technologies in SQL Database.

## In this section

This section provides includes the following topics:

| TOPIC | DESCRIPTION |
| --- | --- |
| Quick Start 1: In-Memory OLTP Technologies for Faster Transact-SQL Performance | Delve right into In-Memory OLTP |

| TOPIC | DESCRIPTION |
| --- | --- |
| Overview and Usage Scenarios | Overview of what In-Memory OLTP is, and what are the scenarios that see performance benefits. |
| Requirements for Using Memory-Optimized Tables | Discusses hardware and software requirements and guidelines for using memory-optimized tables. |
| In-Memory OLTP Code Samples | Contains code samples that show how to create and use a memory-optimized table. |
| Memory-Optimized Tables | Introduces memory-optimized tables. |
| Memory-Optimized Table Variables | Code example showing how to use a memory-optimized table variable instead of a traditional table variable to reduce tempdb use. |
| Indexes on Memory-Optimized Tables | Introduces memory-optimized indexes. |
| Natively Compiled Stored Procedures | Introduces natively compiled stored procedures. |
| Managing Memory for In-Memory OLTP | Understanding and managing memory usage on your system. |
| Creating and Managing Storage for Memory-Optimized Objects | Discusses data and delta files, which store information about transactions in memory-optimized tables. |
| Backup, Restore, and Recovery of Memory-Optimized Tables | Discusses backup, restore, and recovery for memory-optimized tables. |
| Transact-SQL Support for In-Memory OLTP | Discusses Transact-SQL support for In-Memory OLTP. |
| High Availability Support for In-Memory OLTP databases | Discusses availability groups and failover clustering in In-Memory OLTP. |
| SQL Server Support for In-Memory OLTP | Lists new and updated syntax and features supporting memory-optimized tables. |
| Migrating to In-Memory OLTP | Discusses how to migrate disk-based tables to memory-optimized tables. |

More information about In-Memory OLTP is available on:

- Video explaining In-Memory OLTP and demonstrating performance benefits.
- In-Memory OLTP Performance Demo v1.0
- SQL Server In-Memory OLTP Internals Technical Whitepaper
- SQL Server In-Memory OLTP and Columnstore Feature Comparison
- What's new for In-Memory OLTP in SQL Server 2016 Part 1 and Part 2
- In-Memory OLTP – Common Workload Patterns and Migration Considerations
- In-Memory OLTP Blog

## See Also

# Indexes

5/3/2018 • 3 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

## Available index types

The following table lists the types of indexes available in SQL Server and provides links to additional information.

| INDEX TYPE | DESCRIPTION | ADDITIONAL INFORMATION |
|---|---|---|
| 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. | Guidelines for Using Indexes on Memory-Optimized Tables<br><br>Hash Index Design Guidelines |
| 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 | Guidelines for Using Indexes on Memory-Optimized Tables<br><br>Memory-Optimized Nonclustered Index Design Guidelines |
| 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. | Clustered and Nonclustered Indexes Described<br><br>Create Clustered Indexes<br><br>Clustered Index Design Guidelines |
| 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. | Clustered and Nonclustered Indexes Described<br><br>Create Nonclustered Indexes<br><br>Nonclustered Index Design Guidelines |
| 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.<br><br>Uniqueness can be a property of both clustered and nonclustered indexes. | Create Unique Indexes<br><br>Unique Index Design Guidelines |

| INDEX TYPE | DESCRIPTION | ADDITIONAL INFORMATION |
|---|---|---|
| Columnstore | An in-memory columnstore index stores and manages data by using column-based data storage and column-based query processing.<br><br>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. | Columnstore Indexes Guide<br><br>Columnstore Index Design Guidelines |
| Index with included columns | A nonclustered index that is extended to include nonkey columns in addition to the key columns. | Create Indexes with Included 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. | Indexes on Computed Columns |
| 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. | Create Filtered Indexes<br><br>Filtered Index Design Guidelines |
| 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. | Spatial Indexes Overview |
| XML | A shredded, and persisted, representation of the XML binary large objects (BLOBs) in the **xml** data type column. | XML Indexes (SQL Server) |
| 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. | Populate Full-Text Indexes |

# Related Content

SQL Server Index Design Guide SORT_IN_TEMPDB Option For Indexes

Disable Indexes and Constraints

Enable Indexes and Constraints

Rename Indexes

Set Index Options

Disk Space Requirements for Index DDL Operations

Reorganize and Rebuild Indexes

Specify Fill Factor for an Index

Pages and Extents Architecture Guide Clustered and Nonclustered Indexes Described

# JSON data in SQL Server

5/3/2018 • 14 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2016) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

JSON is a popular textual data format that's used for exchanging data in modern web and mobile applications. JSON is also used for storing unstructured data in log files or NoSQL databases such as Microsoft Azure Cosmos DB. Many REST web services return results that are formatted as JSON text or accept data that's formatted as JSON. For example, most Azure services, such as Azure Search, Azure Storage, and Azure Cosmos DB, have REST endpoints that return or consume JSON. JSON is also the main format for exchanging data between webpages and web servers by using AJAX calls.

JSON functions in SQL Server enable you to combine NoSQL and relational concepts in the same database. Now you can combine classic relational columns with columns that contain documents formatted as JSON text in the same table, parse and import JSON documents in relational structures, or format relational data to JSON text. You see see how JSON functions connect relational and NoSQL concepts in SQL Server and Azure SQL Database in the following video:

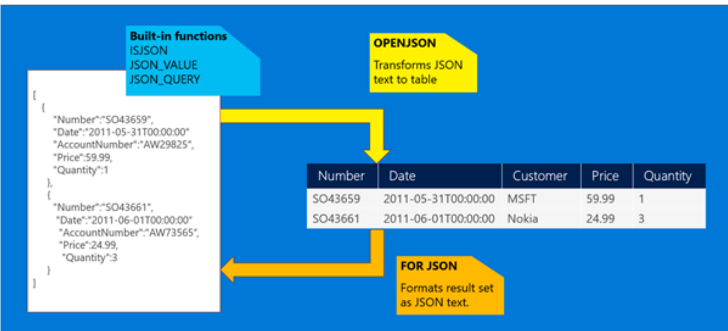
*JSON as a bridge between NoSQL and relational worlds*

Here's an example of JSON text:

```
[{
    "name": "John",
    "skills": ["SQL", "C#", "Azure"]
}, {
    "name": "Jane",
    "surname": "Doe"
}]
```

By using SQL Server built-in functions and operators, you can do the following things with JSON text:

- Parse JSON text and read or modify values.
- Transform arrays of JSON objects into table format.
- Run any Transact-SQL query on the converted JSON objects.
- Format the results of Transact-SQL queries in JSON format.




## Key JSON capabilities of SQL Server and SQL Database

The next sections discuss the key capabilities that SQL Server provides with its built-in JSON support. You can see how to use JSON functions and operators in the following video:

*SQL Server 2016 and JSON Support*

#### Extract values from JSON text and use them in queries

If you have JSON text that's stored in database tables, you can read or modify values in the JSON text by using the following built-in functions:

- ISJSON (Transact-SQL) extracts a scalar value from a JSON string.
- JSON_VALUE (Transact-SQL) extracts an object or an array from a JSON string.
- JSON_QUERY (Transact-SQL) tests whether a string contains valid JSON.
- JSON_MODIFY (Transact-SQL) changes a value in a JSON string.

**Example**

In the following example, the query uses both relational and JSON data (stored in a column named `jsonCol`) from a table:

```
SELECT Name,Surname,
 JSON_VALUE(jsonCol,'$.info.address.PostCode') AS PostCode,
 JSON_VALUE(jsonCol,'$.info.address."Address Line 1"')+' '
  +JSON_VALUE(jsonCol,'$.info.address."Address Line 2"') AS Address,
 JSON_QUERY(jsonCol,'$.info.skills') AS Skills
FROM People
WHERE ISJSON(jsonCol)>0
 AND JSON_VALUE(jsonCol,'$.info.address.Town')='Belgrade'
 AND Status='Active'
ORDER BY JSON_VALUE(jsonCol,'$.info.address.PostCode')
```

Applications and tools see no difference between the values taken from scalar table columns and the values taken from JSON columns. You can use values from JSON text in any part of a Transact-SQL query (including WHERE, ORDER BY, or GROUP BY clauses, window aggregates, and so on). JSON functions use JavaScript-like syntax for referencing values inside JSON text.

For more information, see Validate, query, and change JSON data with built-in functions (SQL Server), JSON_VALUE (Transact-SQL), and JSON_QUERY (Transact-SQL).

### Change JSON values

If you must modify parts of JSON text, you can use the JSON_MODIFY (Transact-SQL) function to update the value of a property in a JSON string and return the updated JSON string. The following example updates the value of a property in a variable that contains JSON:

```
DECLARE @json NVARCHAR(MAX);
SET @json = '{"info":{"address":[{"town":"Belgrade"},{"town":"Paris"},{"town":"Madrid"}]}';
SET @json = JSON_MODIFY(@jsonInfo,'$.info.address[1].town','London');
SELECT modifiedJson = @json;
```

**Results**

| MODIFIEDJSON |
| --- |
| {"info":{"address":[{"town":"Belgrade"},{"town":"London"},{"town":"Madrid"}]} |

### Convert JSON collections to a rowset

You don't need a custom query language to query JSON in SQL Server. To query JSON data, you can use standard T-SQL. If you must create a query or report on JSON data, you can easily convert JSON data to rows and columns by calling the **OPENJSON** rowset function. For more information, see Convert JSON Data to Rows and Columns with OPENJSON (SQL Server).

The following example calls **OPENJSON** and transforms the array of objects that is stored in the `@json` variable to a rowset that can be queried with a standard SQL **SELECT** statement:

```
DECLARE @json NVARCHAR(MAX)
SET @json =
N'[
       { "id" : 2,"info": { "name": "John", "surname": "Smith" }, "age": 25 },
       { "id" : 5,"info": { "name": "Jane", "surname": "Smith" }, "dob": "2005-11-04T12:00:00" }
 ]'

SELECT *
FROM OPENJSON(@json)
  WITH (id int 'strict $.id',
        firstName nvarchar(50) '$.info.name', lastName nvarchar(50) '$.info.surname',
        age int, dateOfBirth datetime2 '$.dob')
```

**Results**

| ID | FIRSTNAME | LASTNAME | AGE | DATEOFBIRTH |
| --- | --- | --- | --- | --- |
| 2 | John | Smith | 25 | |
| 5 | Jane | Smith | | 2005-11-04T12:00:00 |

**OPENJSON** transforms the array of JSON objects into a table in which each object is represented as one row, and key/value pairs are returned as cells. The output observes the following rules:

- **OPENJSON** converts JSON values to the types that are specified in the **WITH** clause.
- **OPENJSON** can handle both flat key/value pairs and nested, hierarchically organized objects.
- You don't have to return all the fields that are contained in the JSON text.
- If JSON values don't exist, **OPENJSON** returns NULL values.
- You can optionally specify a path after the type specification to reference a nested property or to reference a property by a different name.
- The optional **strict** prefix in the path specifies that values for the specified properties must exist in the JSON text.

For more information, see Convert JSON Data to Rows and Columns with OPENJSON (SQL Server) and OPENJSON (Transact-SQL).

JSON documents may have sub-elements and hierarchical data that cannot be directly mapped into the standard relational columns. In this case, you can flatten JSON hierarchy by joining parent entity with sub-arrays.

In the following example, the second object in the array has sub-array representing person skills. Every sub-object can be parsed using additional `OPENJSON` function call:

```
DECLARE @json NVARCHAR(MAX)
SET @json =
N'[
       { "id" : 2,"info": { "name": "John", "surname": "Smith" }, "age": 25 },
       { "id" : 5,"info": { "name": "Jane", "surname": "Smith", "skills": ["SQL", "C#", "Azure"] }, "dob": "2005-11-04T12:00:00" }
 ]'

SELECT *
FROM OPENJSON(@json)
  WITH (id int 'strict $.id',
        firstName nvarchar(50) '$.info.name', lastName nvarchar(50) '$.info.surname',
        age int, dateOfBirth datetime2 '$.dob',
    skills nvarchar(max) '$.skills' as json)
    outer apply openjson( a.skills )
                 with ( skill nvarchar(8) '$' ) as b
```

**skills** array is returned in the first `OPENJSON` as original JSON text fragment and passed to another `OPENJSON` function using `APPLY` operator. The second `OPENJSON` function will parse JSON array and return string values as single column rowset that will be joined with the result of the first `OPENJSON`. The result of this query is shown in the following table:

**Results**

| ID | FIRSTNAME | LASTNAME | AGE | DATEOFBIRTH | SKILL |
|---|---|---|---|---|---|
| 2 | John | Smith | 25 | | |
| 5 | Jane | Smith | | 2005-11-04T12:00:00 | SQL |
| 5 | Jane | Smith | | 2005-11-04T12:00:00 | C# |
| 5 | Jane | Smith | | 2005-11-04T12:00:00 | Azure |

`OUTER APPLY OPENJSON` will join first level entity with sub-array and return flatten resultset. Due to JOIN, the second row will be repeated for every skill.

**Convert SQL Server data to JSON or export JSON**

Format SQL Server data or the results of SQL queries as JSON by adding the **FOR JSON** clause to a **SELECT** statement. Use **FOR JSON** to delegate the formatting of JSON output from your client applications to SQL Server. For more information, see Format Query Results as JSON with FOR JSON (SQL Server).

The following example uses PATH mode with the **FOR JSON** clause:

```
SELECT id, firstName AS "info.name", lastName AS "info.surname", age, dateOfBirth as dob
FROM People
FOR JSON PATH
```

The **FOR JSON** clause formats SQL results as JSON text that can be provided to any app that understands JSON. The PATH option uses dot-separated aliases in the SELECT clause to nest objects in the query results.

**Results**

```
[{
    "id": 2,
    "info": {
        "name": "John",
        "surname": "Smith"
    },
    "age": 25
}, {
    "id": 5,
    "info": {
        "name": "Jane",
        "surname": "Smith"
    },
    "dob": "2005-11-04T12:00:00"
}]
```

For more information, see Format query results as JSON with FOR JSON (SQL Server) and FOR Clause (Transact-SQL).

# Use cases for JSON data in SQL Server

JSON support in SQL Server and Azure SQL Database lets you combine relational and NoSQL concepts. You can easily transform relational to semi-

structured data and vice-versa. JSON is not a replacement for existing relational models, however. Here are some specific use cases that benefit from the JSON support in SQL Server and in SQL Database. For more info, see JSON in SQL Server – Use cases.

#### Simplify complex data models

Consider denormalizing your data model with JSON fields in place of multiple child tables. For more info, see Simplify data access using de-normalized models.

#### Store retail and e-commerce data

Store info about products with a wide range of variable attributes in a denormalized model for flexibility. For more info, see Designing Product Catalogs in SQL Server using JSON and Indexing data in JSON product catalogs.

#### Process log and telemetry data

Load, query, and analyze log data stored as JSON files with all the power of the Transact-SQL language. For more info, see the section *Log and telemetry data analysis* in JSON in SQL Server – Use cases.

#### Store semi-structured IoT data

When you need real-time analysis of IoT data, load the incoming data directly into the database instead of staging it in a storage location. For more info, see Working with Azure IoT data in Azure SQL Database.

#### Simplify REST API development

Transform relational data from your database easily into the JSON format used by the REST APIs that support your web site. For more info, see Simplify REST API development for modern Single-page apps with SQL Server.

## Combine relational and JSON data

SQL Server provides a hybrid model for storing and processing both relational and JSON data by using standard Transact-SQL language. You can organize collections of your JSON documents in tables, establish relationships between them, combine strongly typed scalar columns stored in tables with flexible key/value pairs stored in JSON columns, and query both scalar and JSON values in one or more tables by using full Transact-SQL.

JSON text is stored in varchar or nvarchar columns and is indexed as plain text. Any SQL Server feature or component that supports text supports JSON, so there are almost no constraints on interaction between JSON and other SQL Server features. You can store JSON in In-memory or Temporal tables, apply Row-Level Security predicates on JSON text, and so on.

If you have pure JSON workloads where you want to use a query language that's customized for the processing of JSON documents, consider Microsoft Azure Cosmos DB.

Here are some use cases that show how you can use the built-in JSON support in SQL Server.

## Store and index JSON data in SQL Server

JSON is a textual format so the JSON documents are can stored in `NVARCHAR` columns in SQL Database. Since `NVARCHAR` type is supported in all SQL Server sub-systems you can put JSON documents in tables with **CLUSTERED COLUMNSTORE** indexes, **memory optimized** tables, or external files that can be read using OPENROWSET or Polybase.

To learn more about your options for storing, indexing, and optimizing JSON data in SQL Server, see the following articles:

- Store JSON documents in SQL Server or SQL Database
- Index JSON data
- Optimize JSON processing with in-memory OLTP

#### Load JSON files into SQL Server

You can format information that's stored in files as standard JSON or line-delimited JSON. SQL Server can import the contents of JSON files, parse it by using the **OPENJSON** or **JSON_VALUE** functions, and load it into tables.

- If your JSON documents are stored in local files, on shared network drives, or in Azure Files locations that can be accessed by SQL Server, you can use bulk import to load your JSON data into SQL Server. For more information about this scenario, see Importing JSON files into SQL Server using OPENROWSET (BULK).
- If your line-delimited JSON files are stored in Azure Blob storage or the Hadoop file system, you can use PolyBase to load JSON text, parse it in Transact-SQL code, and load it into tables.

#### Import JSON data into SQL Server tables

If you must load JSON data from an external service into SQL Server, you can use **OPENJSON** to import the data into SQL Server instead of parsing the data in the application layer.

```
DECLARE @jsonVariable NVARCHAR(MAX)

SET @jsonVariable = N'[
        {
          "Order": {
            "Number":"SO43659",
            "Date":"2011-05-31T00:00:00"
          },
          "AccountNumber":"AW29825",
          "Item": {
            "Price":2024.9940,
            "Quantity":1
          }
        },
        {
          "Order": {
            "Number":"SO43661",
            "Date":"2011-06-01T00:00:00"
          },
          "AccountNumber":"AW73565",
          "Item": {
            "Price":2024.9940,
            "Quantity":3
          }
       }
  ]'

INSERT INTO SalesReport
SELECT SalesOrderJsonData.*
FROM OPENJSON (@jsonVariable, N'$.Orders.OrdersArray')
           WITH (
              Number   varchar(200) N'$.Order.Number',
              Date     datetime     N'$.Order.Date',
              Customer varchar(200) N'$.AccountNumber',
              Quantity int          N'$.Item.Quantity'
           )
  AS SalesOrderJsonData;
```

You can provide the content of the JSON variable by an external REST service, send it as a parameter from a client-side JavaScript framework, or load it from external files. You can easily insert, update, or merge results from JSON text into a SQL Server table. For more information about this scenario, see the following blog posts:

- Import JSON data in SQL Server
- Upsert JSON documents in SQL Server 2016
- Load GeoJSON data into SQL Server 2016

## Analyze JSON data with SQL queries

If you must filter or aggregate JSON data for reporting purposes, you can use **OPENJSON** to transform JSON to relational format. You can then use standard Transact-SQL and built-in functions to prepare the reports.

```
SELECT Tab.Id, SalesOrderJsonData.Customer, SalesOrderJsonData.Date
FROM   SalesOrderRecord AS Tab
          CROSS APPLY
     OPENJSON (Tab.json, N'$.Orders.OrdersArray')
           WITH (
              Number   varchar(200) N'$.Order.Number',
              Date     datetime     N'$.Order.Date',
              Customer varchar(200) N'$.AccountNumber',
              Quantity int          N'$.Item.Quantity'
           )
  AS SalesOrderJsonData
WHERE JSON_VALUE(Tab.json, '$.Status') = N'Closed'
ORDER BY JSON_VALUE(Tab.json, '$.Group'), Tab.DateModified
```

You can use both standard table columns and values from JSON text in the same query. You can add indexes on the `JSON_VALUE(Tab.json, '$.Status')` expression to improve the performance of the query. For more information, see Index JSON data.

## Return data from a SQL Server table formatted as JSON

If you have a web service that takes data from the database layer and returns it in JSON format, or if you have JavaScript frameworks or libraries that accept data formatted as JSON, you can format JSON output directly in a SQL query. Instead of writing code or including a library to convert tabular query results and then serialize objects to JSON format, you can use **FOR JSON** to delegate the JSON formatting to SQL Server.

For example, you might want to generate JSON output that's compliant with the OData specification. The web service expects a request and response in the following format:

- Request: `/Northwind/Northwind.svc/Products(1)?$select=ProductID,ProductName`
- Response:

```
{"@odata.context":"http://services.odata.org/V4/Northwind/Northwind.svc/$metadata#Products(ProductID,ProductName)/$entity","ProductID":1,"ProductName":"Chai"
```

This OData URL represents a request for the ProductID and ProductName columns for the product with `id` 1. You can use **FOR JSON** to format the output as expected in SQL Server.

```
SELECT 'http://services.odata.org/V4/Northwind/Northwind.svc/$metadata#Products(ProductID,ProductName)/$entity'
 AS '@odata.context',
 ProductID, Name as ProductName
FROM Production.Product
WHERE ProductID = 1
FOR JSON AUTO
```

The output of this query is JSON text that's fully compliant with the OData spec. Formatting and escaping are handled by SQL Server. SQL Server can also format query results in any format, such as OData JSON or GeoJSON. For more information, see Returning spatial data in GeoJSON format.

## Test drive built-in JSON support with the AdventureWorks sample database

To get the AdventureWorks sample database, download at least the database file and the samples and scripts file from Microsoft Download Center.

After you restore the sample database to an instance of SQL Server 2016, extract the samples file, and then open the *JSON Sample Queries procedures views and indexes.sql* file from the JSON folder. Run the scripts in this file to reformat some existing data as JSON data, test sample queries and reports over the JSON data, index the JSON data, and import and export JSON.

Here's what you can do with the scripts that are included in the file:

- Denormalize the existing schema to create columns of JSON data.
  - Store information from SalesReasons, SalesOrderDetails, SalesPerson, Customer, and other tables that contain information related to sales order into JSON columns in the SalesOrder_json table.
  - Store information from EmailAddresses/PersonPhone tables in the Person_json table as arrays of JSON objects.
- Create procedures and views that query JSON data.
- Index JSON data. Create indexes on JSON properties and full-text indexes.
- Import and export JSON. Create and run procedures that export the content of the Person and the SalesOrder tables as JSON results, and import and update the Person and the SalesOrder tables by using JSON input.
- Run query examples. Run some queries that call the stored procedures and views that you created in steps 2 and 4.
- Clean up scripts. Don't run this part if you want to keep the stored procedures and views that you created in steps 2 and 4.

## Learn more about JSON in SQL Server and Azure SQL Database

### Microsoft blog posts

For specific solutions, use cases, and recommendations, see these blog posts about the built-in JSON support in SQL Server and Azure SQL Database.

### Microsoft videos

For a visual introduction to the built-in JSON support in SQL Server and Azure SQL Database, see the following video:

*Using JSON in SQL Server 2016 and Azure SQL Database*

*Building REST API with SQL Server using JSON functions*

### Reference articles

- FOR Clause (Transact-SQL) (FOR JSON)
- OPENJSON (Transact-SQL)
- JSON Functions (Transact-SQL)
  - ISJSON (Transact-SQL)
  - JSON_VALUE (Transact-SQL)
  - JSON_QUERY (Transact-SQL)
  - JSON_MODIFY (Transact-SQL)

# Linked Servers (Database Engine)

5/3/2018 • 3 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Configure a linked server to enable the SQL Server Database Engine to execute commands against OLE DB data sources outside of the instance of SQL Server. Typically linked servers are configured to enable the Database Engine to execute a Transact-SQL statement that includes tables in another instance of SQL Server, or another database product such as Oracle. Many types OLE DB data sources can be configured as linked servers, including Microsoft Access and Excel. Linked servers offer the following advantages:

- The ability to access data from outside of SQL Server.
- The ability to issue distributed queries, updates, commands, and transactions on heterogeneous data sources across the enterprise.
- The ability to address diverse data sources similarly.

  You can configure a linked server by using SQL Server Management Studio or by using the sp_addlinkedserver (Transact-SQL) statement. OLE DB providers vary greatly in the type and number of parameters required. For example some providers require you to provide a security context for the connection using sp_addlinkedsrvlogin (Transact-SQL). Some OLE DB providers allow SQL Server to update data on the OLE DB source. Others provide only read-only data access. For information about each OLE DB provider, consult documentation for that OLE DB provider.

## Linked Server Components

A linked server definition specifies the following objects:

- An OLE DB provider
- An OLE DB data source

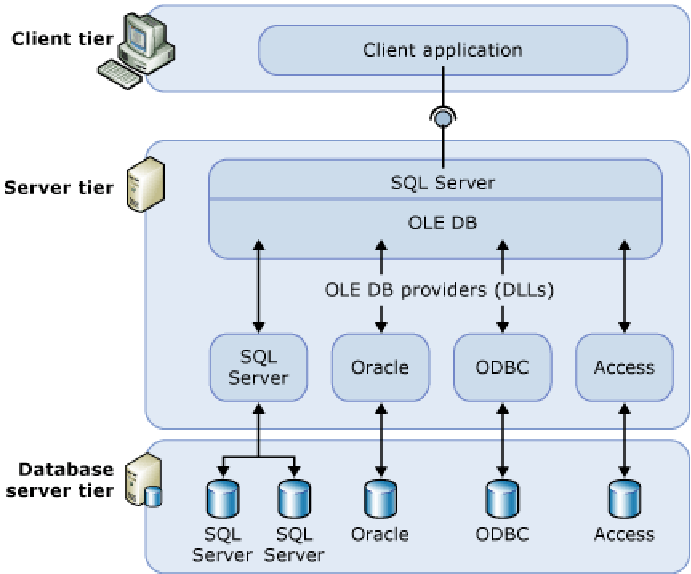
  An *OLE DB provider* is a DLL that manages and interacts with a specific data source. An *OLE DB data source* identifies the specific database that can be accessed through OLE DB. Although data sources queried through linked server definitions are ordinarily databases, OLE DB providers exist for a variety of files and file formats. These include text files, spreadsheet data, and the results of full-text content searches.

  The Microsoft SQL Server Native Client OLE DB Provider (PROGID: SQLNCLI11) is the official OLE DB provider for SQL Server.

> **NOTE**
>
> SQL Server distributed queries are designed to work with any OLE DB provider that implements the required OLE DB interfaces. However, SQL Server has been tested against only the SQL Server Native Client OLE DB Provider and certain other providers.

## Linked Server Details

The following illustration shows the basics of a linked server configuration.




Typically, linked servers are used to handle distributed queries. When a client application executes a distributed query through a linked server, SQL Server parses the command and sends requests to OLE DB. The rowset request may be in the form of executing a query against the provider or opening a base table from the provider.

For a data source to return data through a linked server, the OLE DB provider (DLL) for that data source must be present on the same server as the instance of SQL Server.

When a third-party OLE DB provider is used, the account under which the SQL Server service runs must have read and execute permissions for the directory, and all subdirectories, in which the provider is installed.

## Managing Providers

There is a set of options that control how SQL Server loads and uses OLE DB providers that are specified in the registry.

## Managing Linked Server Definitions

When you are setting up a linked server, register the connection information and data source information with SQL Server. After registered, that data source can be referred to with a single logical name.

You can use stored procedures and catalog views to manage linked server definitions:

- Create a linked server definition by running **sp_addlinkedserver**.
- View information about the linked servers defined in a specific instance of SQL Server by running a query against the **sys.servers** system catalog views.
- Delete a linked server definition by running **sp_dropserver**. You can also use this stored procedure to remove a remote server.

  You can also define linked servers by using SQL Server Management Studio. In the Object Explorer, right-click **Server Objects**, select **New**, and select **Linked Server**. You can delete a linked server definition by right-clicking the linked server name and selecting **Delete**.

  When you execute a distributed query against a linked server, include a fully qualified, four-part table name for each data source to query. This four-part name should be in the form *linked_server_name.catalog.schema.object_name*.

> **NOTE**
>
> Linked servers can be defined to point back (loop back) to the server on which they are defined. Loopback servers are most useful when testing an application that uses distributed queries on a single server network. Loopback linked servers are intended for testing and are not supported for many operations, such as distributed transactions.

## Related Tasks

Create Linked Servers (SQL Server Database Engine)

sp_addlinkedserver (Transact-SQL)

sp_addlinkedsrvlogin (Transact-SQL)

sp_dropserver (Transact-SQL)

## Related Content

sys.servers (Transact-SQL)

sp_linkedservers (Transact-SQL)

# Maintenance Plans

5/3/2018 • 2 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Maintenance plans create a workflow of the tasks required to make sure that your database is optimized, regularly backed up, and free of inconsistencies. The Maintenance Plan Wizard also creates core maintenance plans, but creating plans manually gives you much more flexibility.

## Benefits of Maintenance Plans

In SQL Server 2017 Database Engine, maintenance plans create an Integration Services package, which is run by a SQL Server Agent job. Maintenance plans can be run manually or automatically at scheduled intervals.

SQL Server 2017 maintenance plans provide the following features:

- Workflow creation using a variety of typical maintenance tasks. You can also create your own custom Transact-SQL scripts.
- Conceptual hierarchies. Each plan lets you create or edit task workflows. Tasks in each plan can be grouped into subplans, which can be scheduled to run at different times.
- Support for multiserver plans that can be used in master server/target server environments.
- Support for logging plan history to remote servers.
- Support for Windows Authentication and SQL Server Authentication. When possible, use Windows Authentication.

## Maintenance Plan Functionality

Maintenance plans can be created to perform the following tasks:

- Reorganize the data on the data and index pages by rebuilding indexes with a new fill factor. Rebuilding indexes with a new fill factor makes sure that database pages contain an equally distributed amount of data and free space. It also enables faster growth in the future. For more information, see Specify Fill Factor for an Index.
- Compress data files by removing empty database pages.
- Update index statistics to make sure the query optimizer has current information about the distribution of data values in the tables. This enables the query optimizer to make better judgments about the best way to access data, because it has more information about the data stored in the database. Although index statistics are automatically updated by SQL Server periodically, this option can force the statistics to update immediately.
- Perform internal consistency checks of the data and data pages within the database to make sure that a system or software problem has not damaged data.
- Back up the database and transaction log files. Database and log backups can be retained for a specified period. This lets you create a history of backups to be used if you have to restore the database to a time earlier than the last database backup. You can also perform differential backups.
- Run SQL Server Agent jobs. This can be used to create jobs that perform a variety of actions and the

maintenance plans to run those jobs.

The results generated by the maintenance tasks can be written as a report to a text file or to the maintenance plan tables (**sysmaintplan_log** and **sysmaintplan_logdetail**) in **msdb**. To view the results in the log file viewer, right-click **Maintenance Plans**, and then click **View History**.

## Related Tasks

Use the following topics to get started with maintenance plans.

| | |
|---|---|
| **Description** | **Topic** |
| Configure the **Agent XPs** server configuration option to enable the SQL Server Agent extended stored procedures. | Agent XPs Server Configuration Option |
| Describes how to create a maintenance plan by using SQL Server Management Studio or Transact-SQL. | Create a Maintenance Plan |
| Describes how to create a maintenance plan by using the Maintenance Plan Design Surface. | Create a Maintenance Plan (Maintenance Plan Design Surface) |
| Documents maintenance plan functionality available in Object Explorer. | Maintenance Plans Node (Object Explorer) |

# SQL Server Utility Features and Tasks

5/3/2018 • 3 min to read • Edit Online

**THIS TOPIC APPLIES TO:** ✅ SQL Server ⊗ Azure SQL Database ⊗ Azure SQL Data Warehouse ⊗ Parallel Data Warehouse

SQL Server customers have a requirement to manage their SQL Server environment as a whole, addressed in this release through the concept of application and multiserver management in the SQL Server Utility.

## Benefits of the SQL Server Utility

The SQL Server Utility models an organization's SQL Server-related entities in a unified view. Utility Explorer and SQL Server Utility viewpoints in SQL Server Management Studio (SSMS) provide administrators a holistic view of SQL Server resource health through an instance of SQL Server that serves as a utility control point (UCP). The combination of summary and detailed data presented in the UCP for both underutilization and overutilization policies, and for a variety of key parameters, enables resource consolidation opportunities and resource overutilization to be identified with ease. Health policies are configurable, and can be adjusted to change either upper or lower resource utilization thresholds. You can change global monitoring policies, or configure individual monitoring policies for each entity managed in the SQL Server Utility.

## Getting Started with SQL Server Utility

The typical user scenario begins with creation of a utility control point which establishes the central reasoning point for the SQL Server Utility. The UCP provides a consolidated view of resource health collected from managed instances of SQL Server in the SQL Server Utility. After the UCP is created, you enroll instances of SQL Server into the SQL Server Utility so that they can be managed by the UCP.

Each instance of SQL Server and data-tier application managed by the SQL Server Utility can be monitored based on global policy definitions or based on individual policy definitions.

## Related Tasks

Use the following topics to get started with SQL Server utility.

| | |
|---|---|
| **Description** | **Topic** |
| Describes considerations to configure a server to run utility and non-utility collection sets on the same instance of SQL Server. | Considerations for Running Utility and non-Utility Collection Sets on the Same Instance of SQL Server |
| Describes how to create a SQL Server utility control point. | Create a SQL Server Utility Control Point (SQL Server Utility) |
| Describes how to connect to a SQL Server Utility. | Connect to a SQL Server Utility |
| Describes how to enroll an instance of SQL Server with a Utility Control Point. | Enroll an Instance of SQL Server (SQL Server Utility) |
| Describes how to use Utility Explorer to manage the SQL Server utility. | Use Utility Explorer to Manage the SQL Server Utility |

| | |
|---|---|
| Describes how to monitor instances of SQL Server in the SQL Server Utility. | Monitor Instances of SQL Server in the SQL Server Utility |
| Describes how to view resource health policy results. | View Resource Health Policy Results (SQL Server Utility) |
| Describes how to modify a resource health policy definition. | Modify a Resource Health Policy Definition (SQL Server Utility) |
| Describes how to configure your UCP data warehouse. | Configure Your Utility Control Point Data Warehouse (SQL Server Utility) |
| Describes how to configure utility health policies. | Configure Health Policies (SQL Server Utility) |
| Describes how to adjust attenuation in CPU utilization policies. | Reduce Noise in CPU Utilization Policies (SQL Server Utility) |
| Describes how to remove an instance of SQL Server from a UCP. | Remove an Instance of SQL Server from the SQL Server Utility |
| Describes how to change the proxy account for the utility data collector on a managed instance of SQL Server. | Change the Proxy Account for the Utility Collection Set on a Managed Instance of SQL Server (SQL Server Utility) |
| Describes how to move a UCP from one instance of SQL Server to another. | Move a UCP from One Instance of SQL Server to Another (SQL Server Utility) |
| Describes how to remove a UCP. | Remove a Utility Control Point (SQL Server Utility) |
| Describes how to troubleshoot the SQL server utility. | Troubleshoot the SQL Server Utility |
| Describes how to troubleshoot SQL Server resource health. | Troubleshoot SQL Server Resource Health (SQL Server Utility) |
| Links to UtilityExplorer F1 Help topics. | Utility Explorer F1 Help |

# Database Lifecycle Management

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Database lifecycle management (DLM) is a policy-based approach to managing databases and data assets. DLM is not a product but a comprehensive approach to managing the database schema, data, and metadata for a database application. A thoughtful and proactive approach to DLM enables an organization to manage data resources according to appropriate levels of performance, protection, availability, and cost.

DLM begins with discussion of project design and intent, continues with database develop, test, build, deploy, maintain, monitor, and backup activities, and ends with data archive. This topic provides an overview of the stages of DLM that begin with database development and progress through build, deploy, and monitor actions (Figure 1). Also included are data management activities, and data portability operations like import/export, backup, migrate, and sync.

To read the complete topic, see Database Lifecycle Management (DLM).

## See Also

Windows Azure Home Page
Windows Azure Developer Center
Windows Azure Manage Center
Windows Azure Team Blog
Windows Azure Support Options

# Administer Multiple Servers Using Central Management Servers

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

You can administer multiple servers by designating Central Management Servers and creating server groups.

## What is a Central Management Server and server groups?

An instance of SQL Server designated as a Central Management Server maintains server groups that contain the connection information for one or more instances. You can execute Transact-SQL statements and Policy-Based Management policies at the same time against server groups. You can also view the log files on instances managed through a Central Management Server.

Basically a Central management Server is central repository containing a list of your managed servers. Versions earlier than SQL Server 2008 cannot be designated as a Central Management Server.

Transact-SQL statements can also be executed against local server groups in Registered Servers.

## Create Central Management Server and server groups

To create a Central Management Server and server groups, use the **Registered Servers** window in SQL Server Management Studio. Note that the Central Management Server cannot be a member of a group that it maintains.

For how to create Central Management Servers and server groups, see Create a Central Management Server and Server Group (SQL Server Management Studio).

## See also

Administer Servers by Using Policy-Based Management

# Joins (SQL Server)

5/3/2018 • 13 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

SQL Server performs sort, intersect, union, and difference operations using in-memory sorting and hash join technology. Using this type of query plan, SQL Server supports vertical table partitioning, sometimes called columnar storage.

SQL Server employs three types of join operations:

- Nested Loops joins
- Merge joins
- Hash joins

## Join Fundamentals

By using joins, you can retrieve data from two or more tables based on logical relationships between the tables. Joins indicate how Microsoft SQL Server should use data from one table to select the rows in another table.

A join condition defines the way two tables are related in a query by:

- Specifying the column from each table to be used for the join. A typical join condition specifies a foreign key from one table and its associated key in the other table.
- Specifying a logical operator (for example, = or <>,) to be used in comparing values from the columns.

Inner joins can be specified in either the `FROM` or `WHERE` clauses. Outer joins can be specified in the `FROM` clause only. The join conditions combine with the `WHERE` and `HAVING` search conditions to control the rows that are selected from the base tables referenced in the `FROM` clause.

Specifying the join conditions in the `FROM` clause helps separate them from any other search conditions that may be specified in a `WHERE` clause, and is the recommended method for specifying joins. A simplified ISO FROM clause join syntax is:

```
FROM first_table join_type second_table [ON (join_condition)]
```

*join_type* specifies what kind of join is performed: an inner, outer, or cross join. *join_condition* defines the predicate to be evaluated for each pair of joined rows. The following is an example of a FROM clause join specification:

```
FROM Purchasing.ProductVendor JOIN Purchasing.Vendor
     ON (ProductVendor.BusinessEntityID = Vendor.BusinessEntityID)
```

The following is a simple SELECT statement using this join:

```
SELECT ProductID, Purchasing.Vendor.BusinessEntityID, Name
FROM Purchasing.ProductVendor JOIN Purchasing.Vendor
    ON (Purchasing.ProductVendor.BusinessEntityID = Purchasing.Vendor.BusinessEntityID)
WHERE StandardPrice > $10
  AND Name LIKE N'F%'
GO
```

The select returns the product and supplier information for any combination of parts supplied by a company for which the company name starts with the letter F and the price of the product is more than $10.

When multiple tables are referenced in a single query, all column references must be unambiguous. In the previous example, both the ProductVendor and Vendor table have a column named BusinessEntityID. Any column name that is duplicated between two or more tables referenced in the query must be qualified with the table name. All references to the Vendor columns in the example are qualified.

When a column name is not duplicated in two or more tables used in the query, references to it do not have to be qualified with the table name. This is shown in the previous example. Such a SELECT statement is sometimes difficult to understand because there is nothing to indicate the table that provided each column. The readability of the query is improved if all columns are qualified with their table names. The readability is further improved if table aliases are used, especially when the table names themselves must be qualified with the database and owner names. The following is the same example, except that table aliases have been assigned and the columns qualified with table aliases to improve readability:

```
SELECT pv.ProductID, v.BusinessEntityID, v.Name
FROM Purchasing.ProductVendor AS pv
JOIN Purchasing.Vendor AS v
    ON (pv.BusinessEntityID = v.BusinessEntityID)
WHERE StandardPrice > $10
    AND Name LIKE N'F%';
```

The previous examples specified the join conditions in the FROM clause, which is the preferred method. The following query contains the same join condition specified in the WHERE clause:

```
SELECT pv.ProductID, v.BusinessEntityID, v.Name
FROM Purchasing.ProductVendor AS pv, Purchasing.Vendor AS v
WHERE pv.BusinessEntityID=v.BusinessEntityID
    AND StandardPrice > $10
    AND Name LIKE N'F%';
```

The select list for a join can reference all the columns in the joined tables, or any subset of the columns. The select list is not required to contain columns from every table in the join. For example, in a three-table join, only one table can be used to bridge from one of the other tables to the third table, and none of the columns from the middle table have to be referenced in the select list.

Although join conditions usually have equality comparisons (=), other comparison or relational operators can be specified, as can other predicates. For more information, see Comparison Operators (Transact-SQL) and WHERE (Transact-SQL).

When SQL Server processes joins, the query engine chooses the most efficient method (out of several possibilities) of processing the join. The physical execution of various joins can use many different optimizations and therefore cannot be reliably predicted.

Columns used in a join condition are not required to have the same name or be the same data type. However, if the data types are not identical, they must be compatible, or be types that SQL Server can implicitly convert. If the data types cannot be implicitly converted, the join condition must explicitly convert the data type using the `CAST` function. For more information about implicit and explicit conversions, see Data Type Conversion (Database Engine).

Most queries using a join can be rewritten using a subquery (a query nested within another query), and most subqueries can be rewritten as joins. For more information about subqueries, see Subqueries.

> **NOTE**
>
> Tables cannot be joined directly on ntext, text, or image columns. However, tables can be joined indirectly on ntext, text, or image columns by using `SUBSTRING`.
> For example, `SELECT * FROM t1 JOIN t2 ON SUBSTRING(t1.textcolumn, 1, 20) = SUBSTRING(t2.textcolumn, 1, 20)` performs a two-table inner join on the first 20 characters of each text column in tables t1 and t2.
> In addition, another possibility for comparing ntext or text columns from two tables is to compare the lengths of the columns with a `WHERE` clause, for example: `WHERE DATALENGTH(p1.pr_info) = DATALENGTH(p2.pr_info)`

## Understanding Nested Loops joins

If one join input is small (fewer than 10 rows) and the other join input is fairly large and indexed on its join columns, an index nested loops join is the fastest join operation because they require the least I/O and the fewest comparisons.

The nested loops join, also called *nested iteration*, uses one join input as the outer input table (shown as the top input in the graphical execution plan) and one as the inner (bottom) input table. The outer loop consumes the outer input table row by row. The inner loop, executed for each outer row, searches for matching rows in the inner input table.

In the simplest case, the search scans an entire table or index; this is called a *naive nested loops join*. If the search exploits an index, it is called an *index nested loops join*. If the index is built as part of the query plan (and destroyed upon completion of the query), it is called a *temporary index nested loops join*. All these variants are considered by the Query Optimizer.

A nested loops join is particularly effective if the outer input is small and the inner input is preindexed and large. In many small transactions, such as those affecting only a small set of rows, index nested loops joins are superior to both merge joins and hash joins. In large queries, however, nested loops joins are often not the optimal choice.

## Understanding Merge joins

If the two join inputs are not small but are sorted on their join column (for example, if they were obtained by scanning sorted indexes), a merge join is the fastest join operation. If both join inputs are large and the two inputs are of similar sizes, a merge join with prior sorting and a hash join offer similar performance. However, hash join operations are often much faster if the two input sizes differ significantly from each other.

The merge join requires both inputs to be sorted on the merge columns, which are defined by the equality (ON) clauses of the join predicate. The query optimizer typically scans an index, if one exists on the proper set of columns, or it places a sort operator below the merge join. In rare cases, there may be multiple equality clauses, but the merge columns are taken from only some of the available equality clauses.

Because each input is sorted, the **Merge Join** operator gets a row from each input and compares them. For example, for inner join operations, the rows are returned if they are equal. If they are not equal, the lower-value row is discarded and another row is obtained from that input. This process repeats until all rows have been processed.

The merge join operation may be either a regular or a many-to-many operation. A many-to-many merge join uses a temporary table to store rows. If there are duplicate values from each input, one of the inputs will have to rewind to the start of the duplicates as each duplicate from the other input is processed.

If a residual predicate is present, all rows that satisfy the merge predicate evaluate the residual predicate, and only those rows that satisfy it are returned.

Merge join itself is very fast, but it can be an expensive choice if sort operations are required. However, if the data volume is large and the desired data can be obtained presorted from existing B-tree indexes, merge join is often

the fastest available join algorithm.

## Understanding Hash joins

Hash joins can efficiently process large, unsorted, nonindexed inputs. They are useful for intermediate results in complex queries because:

- Intermediate results are not indexed (unless explicitly saved to disk and then indexed) and often are not suitably sorted for the next operation in the query plan.
- Query optimizers estimate only intermediate result sizes. Because estimates can be very inaccurate for complex queries, algorithms to process intermediate results not only must be efficient, but also must degrade gracefully if an intermediate result turns out to be much larger than anticipated.

The hash join allows reductions in the use of denormalization. Denormalization is typically used to achieve better performance by reducing join operations, in spite of the dangers of redundancy, such as inconsistent updates. Hash joins reduce the need to denormalize. Hash joins allow vertical partitioning (representing groups of columns from a single table in separate files or indexes) to become a viable option for physical database design.

The hash join has two inputs: the **build** input and **probe** input. The query optimizer assigns these roles so that the smaller of the two inputs is the build input.

Hash joins are used for many types of set-matching operations: inner join; left, right, and full outer join; left and right semi-join; intersection; union; and difference. Moreover, a variant of the hash join can do duplicate removal and grouping, such as `SUM(salary) GROUP BY department`. These modifications use only one input for both the build and probe roles.

The following sections describe different types of hash joins: in-memory hash join, grace hash join, and recursive hash join.

#### In-Memory Hash Join

The hash join first scans or computes the entire build input and then builds a hash table in memory. Each row is inserted into a hash bucket depending on the hash value computed for the hash key. If the entire build input is smaller than the available memory, all rows can be inserted into the hash table. This build phase is followed by the probe phase. The entire probe input is scanned or computed one row at a time, and for each probe row, the hash key's value is computed, the corresponding hash bucket is scanned, and the matches are produced.

#### Grace Hash Join

If the build input does not fit in memory, a hash join proceeds in several steps. This is known as a grace hash join. Each step has a build phase and probe phase. Initially, the entire build and probe inputs are consumed and partitioned (using a hash function on the hash keys) into multiple files. Using the hash function on the hash keys guarantees that any two joining records must be in the same pair of files. Therefore, the task of joining two large inputs has been reduced to multiple, but smaller, instances of the same tasks. The hash join is then applied to each pair of partitioned files.

#### Recursive Hash Join

If the build input is so large that inputs for a standard external merge would require multiple merge levels, multiple partitioning steps and multiple partitioning levels are required. If only some of the partitions are large, additional partitioning steps are used for only those specific partitions. In order to make all partitioning steps as fast as possible, large, asynchronous I/O operations are used so that a single thread can keep multiple disk drives busy.

> **NOTE**
>
> If the build input is only slightly larger than the available memory, elements of in-memory hash join and grace hash join are combined in a single step, producing a hybrid hash join.

It is not always possible during optimization to determine which hash join is used. Therefore, SQL Server starts by using an in-memory hash join and gradually transitions to grace hash join, and recursive hash join, depending on the size of the build input.

If the Query Optimizer anticipates wrongly which of the two inputs is smaller and, therefore, should have been the build input, the build and probe roles are reversed dynamically. The hash join makes sure that it uses the smaller overflow file as build input. This technique is called role reversal. Role reversal occurs inside the hash join after at least one spill to the disk.

> **NOTE**
>
> Role reversal occurs independent of any query hints or structure. Role reversal does not display in your query plan; when it occurs, it is transparent to the user.

**Hash Bailout**

The term hash bailout is sometimes used to describe grace hash joins or recursive hash joins.

> **NOTE**
>
> Recursive hash joins or hash bailouts cause reduced performance in your server. If you see many Hash Warning events in a trace, update statistics on the columns that are being joined.

For more information about hash bailout, see Hash Warning Event Class.

## Null Values and Joins

When there are null values in the columns of the tables being joined, the null values do not match each other. The presence of null values in a column from one of the tables being joined can be returned only by using an outer join (unless the `WHERE` clause excludes null values).

Here are two tables that each have NULL in the column that will participate in the join:

```
table1                          table2
a           b                   c             d
-------     ------              -------       ------
      1        one                 NULL          two
   NULL      three                    4         four
      4      join4
```

A join that compares the values in column a against column c does not get a match on the columns that have values of NULL:

```
SELECT *
FROM table1 t1 JOIN table2 t2
   ON t1.a = t2.c
ORDER BY t1.a;
GO
```

Only one row with 4 in column a and c is returned:

```
a           b      c           d
----------- ------ ----------- ------
4           join4  4           four

(1 row(s) affected)
```

Null values returned from a base table are also difficult to distinguish from the null values returned from an outer join. For example, the following `SELECT` statement does a left outer join on these two tables:

```
SELECT *
FROM table1 t1 LEFT OUTER JOIN table2 t2
   ON t1.a = t2.c
ORDER BY t1.a;
GO
```

Here is the result set.

```
a           b      c           d
----------- ------ ----------- ------
NULL        three  NULL        NULL
1           one    NULL        NULL
4           join4  4           four

(3 row(s) affected)
```

The results do not make it easy to distinguish a NULL in the data from a NULL that represents a failure to join. When null values are present in data being joined, it is usually preferable to omit them from the results by using a regular join.

## See Also

Showplan Logical and Physical Operators Reference
Comparison Operators (Transact-SQL)
Data Type Conversion (Database Engine)
Subqueries
Adaptive Joins

# Partitioned Tables and Indexes

5/3/2018 • 10 min to read • Edit Online

**THIS TOPIC APPLIES TO:** ✅ SQL Server ✅ Azure SQL Database ❌ Azure SQL Data Warehouse ❌ Parallel Data Warehouse

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. The table or index is treated as a single logical entity when queries or updates are performed on the data. Prior to SQL Server 2016 (13.x) SP1, partitioned tables and indexes were not available in every edition of SQL Server. For a list of features that are supported by the editions of SQL Server, see Editions and Supported Features for SQL Server 2016.

> **IMPORTANT**
>
> SQL Server 2017 supports up to 15,000 partitions by default. In versions earlier than SQL Server 2012 (11.x), 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. Computed columns that participate in a partition function must be explicitly marked PERSISTED. All data types that are valid for use as index columns can be used as a partitioning column, except **timestamp**. The **ntext**, **text**, **image**, **xml**, **varchar(max)**, **nvarchar(max)**, or **varbinary(max)** data types cannot be specified. Also, Microsoft .NET Framework common language runtime (CLR) user-defined type and alias data type columns 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. An index does not have to participate in the same named partition function to be aligned with its base table. However, the partition function of the index and the base table must be essentially the same, in that 1) the arguments of the partition functions have the same data type, 2) they define the same number of partitions, and 3) they define the same boundary values for partitions.

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

The new, higher limit of 15,000 partitions affects memory, partitioned index operations, DBCC commands, and queries. This section describes the performance implications of increasing the number of partitions above 1,000 and provides workarounds as needed. With the limit on the maximum number of partitions being increased to 15,000, you can store data for a longer time. However, you should retain data only for as long as it is needed and maintain a balance between performance and number of partitions.

### 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. Systems with 16 GB of RAM that run many memory-intensive processes may run out of memory on operations that run on a large number of partitions.

Therefore, the more memory you have over 16 GB, the less likely you are to encounter performance and memory issues.

Memory limitations can affect the performance or ability of SQL Server to build a partitioned index. This is especially the case when the index is not aligned with its base table or is not aligned with its clustered index, if the table already has a clustered index applied to it.

**Partitioned Index Operations**

Memory limitations can affect the performance or ability of SQL Server to build a partitioned index. This is especially the case with nonaligned indexes. Creating and rebuilding nonaligned indexes on a table with more than 1,000 partitions is possible, but is not supported. Doing so may cause degraded performance or excessive memory consumption during these 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. For example, a nonaligned partitioned index with 100 partitions requires sufficient memory to serially sort 4,000 (40 * 100) pages at the same time. If this memory is available, the build operation will succeed, but performance may suffer. If this memory is not available, the build operation will fail. Alternatively, an aligned partitioned index with 100 partitions requires only sufficient memory to sort 40 pages, because the sorts are not performed at the same time.

For both aligned and nonaligned indexes, the memory requirement can be greater if SQL Server is applying degrees of parallelism to the build operation on a multiprocessor computer. This is because the greater the degrees of parallelism, the greater the memory requirement. For example, if SQL Server sets degrees of parallelism to 4, a nonaligned partitioned index with 100 partitions requires sufficient memory for four processors to sort 4,000 pages at the same time, or 16,000 pages. If the partitioned index is aligned, the memory requirement is reduced to four processors sorting 40 pages, or 160 (4 * 40) pages. You can use the MAXDOP index option to manually reduce the degrees of parallelism.

**DBCC Commands**

With a larger number of partitions, DBCC commands could take longer to execute as the number of partitions increases.

**Queries**

Queries that use partition elimination could have comparable or improved performance with larger number of partitions. Queries that do not use partition elimination could take longer to execute as the number of partitions increases.

For example, assume a table has 100 million rows and columns `A`, `B`, and `C`. In scenario 1, the table is divided into 1000 partitions on column `A`. In scenario 2, the table is divided into 10,000 partitions on column `A`. A query on the table that has a WHERE clause filtering on column `A` will perform partition elimination and scan one partition. That same query may run faster in scenario 2 as there are fewer rows to scan in a partition. A query that has a WHERE clause filtering on column B will scan all partitions. The query may run faster in scenario 1 than in scenario 2 as there are fewer partitions to scan.

Queries that use operators such as TOP or MAX/MIN on columns other than the partitioning column may

experience reduced performance with partitioning because all partitions must be evaluated.

## Behavior Changes in Statistics Computation During Partitioned Index Operations

Beginning with SQL Server 2012 (11.x), statistics are not created by scanning all the rows in the table when a partitioned index is created or rebuilt. Instead, the query optimizer uses the default sampling algorithm to generate statistics. After upgrading a database with partitioned indexes, you may notice a difference in the histogram data for these indexes. This change in behavior may not affect query performance. To obtain statistics on partitioned indexes by scanning all the rows in the table, use CREATE STATISTICS or UPDATE STATISTICS with the FULLSCAN clause.

## Related Tasks

| Tasks | Topic |
| --- | --- |
| Describes how to create partition functions and partition schemes and then apply these to a table and index. | Create Partitioned Tables and Indexes |

## Related Content

You may find the following white papers on partitioned table and index strategies and implementations useful.

- Partitioned Table and Index Strategies Using SQL Server 2008
- How to Implement an Automatic Sliding Window
- Bulk Loading into a Partitioned Table
- Project REAL: Data Lifecycle -- Partitioning
- Query Processing Enhancements on Partitioned Tables and Indexes
- Top 10 Best Practices for Building a Large Scale Relational Data Warehouse

# Administer Servers by Using Policy-Based Management

5/3/2018 • 5 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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 SQL Server management Studio to create and manage policies, to:

1. Select a Policy-Based Management facet that contains the properties to be configured.
2. Define a condition that specifies the state of a management facet.
3. Define a policy that contains the condition, additional conditions that filter the target sets, and the evaluation mode.
4. Check whether an instance of SQL Server is in compliance with the policy.

   For failed policies, Object Explorer indicates a critical health warning as a red icon next to the target and the nodes that are higher in the Object Explorer tree.

> **NOTE:** When the system computes the object set for a policy, by default the system objects are excluded. For example, if the object set of the policy refers to all tables, the policy will not apply to system tables. If users want to evaluate a policy against system objects, they can explicitly add system objects to the object set. However, though all policies are supported for **check on schedule** evaluation mode, for performance reason, not all policies with arbitrary object sets are supported for **check on change** evaluation mode. For more information, see http://blogs.msdn.com/b/sqlpbm/archive/2009/04/13/policy-evaluation-modes.aspx

## 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.

**Policy-Based Management policy category**
A user-defined category to help manage policies. Users can classify policies into different policy categories. A policy belongs to one and only one policy category. Policy categories apply to databases and servers. At the database level, the following conditions apply:

- Database owners can subscribe a database to a set of policy categories.
- Only policies from its subscribed categories can govern a database.
- All databases implicitly subscribe to the default policy category.

  At the server level, policy categories can be applied to all databases.

  **Effective policy**
  The effective policies of a target are those policies that govern this target. A policy is effective with regard to a target only if all the following conditions are satisfied:

- The policy is enabled.
- The target belongs to the target set of the policy.

- The target or one of the targets ancestors subscribes to the policy group that contains this policy.

## Links to specific tasks

- Store Policy-Based Management policies.|
- Configure Alerts to Notify Policy Administrators of Policy Failures
- Create a New Policy-Based Management Condition
- Delete a Policy-Based Management Condition
- View or Modify the Properties of a Policy-Based Management Condition|
- Export a Policy-Based Management Policy
- Import a Policy-Based Management Policy|
- Evaluate a Policy-Based Management Policy from an Object
- Work with Policy-Based Management Facets|
- Monitor and Enforce Best Practices Using Policy-Based Management

## Examples

- Create the Off By Default Policy
  - Configure a Server to Run the Off By Default Policy ## See also
    Policy-Based Management Views (Transact-SQL)

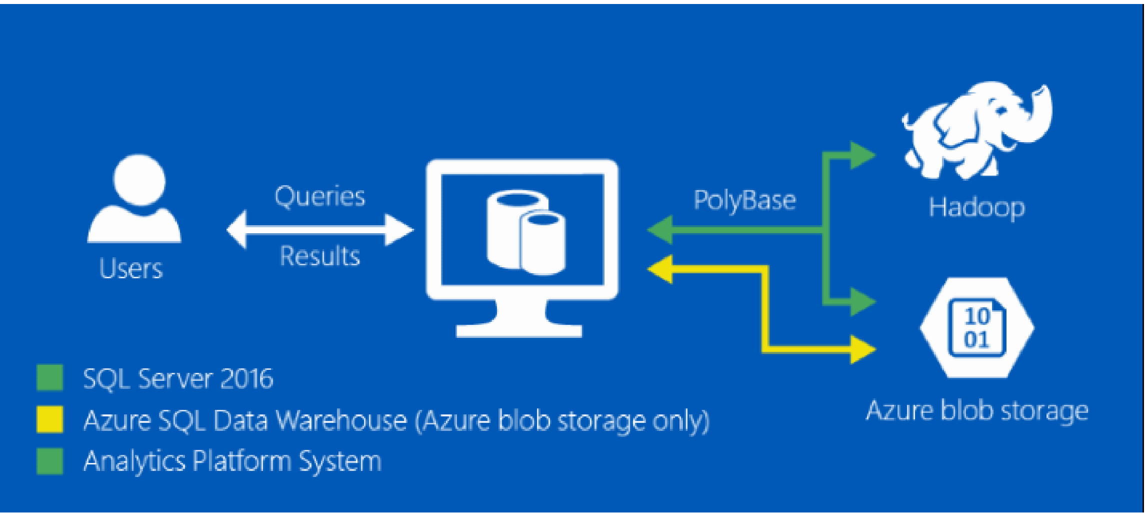
# PolyBase Guide

5/4/2018 • 3 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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.

To use PolyBase, see Get started with PolyBase.




## 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.

## Performance

- **Push computation to Hadoop.**The query optimizer makes a cost-based decision to push computation to Hadoop when doing so will improve query performance. It uses statistics on external tables to make the cost-based decision. Pushing computation creates MapReduce jobs and leverages Hadoop's distributed computational resources.
- **Scale compute resources.** To improve query performance, you can use SQL Server PolyBase scale-out groups. This enables parallel data transfer between SQL Server instances and Hadoop nodes, and it adds compute resources for operating on the external data.

## PolyBase Guide Topics

This guide includes topics to help you use PolyBase efficiently and effectively.

| Topic | Description |
|---|---|
| Get started with PolyBase | Basic steps to install and configure PolyBase. This shows how to create external objects that point to data in Hadoop or Azure blob storage, and gives query examples. |
| PolyBase Versioned Feature Summary | Describes which PolyBase features are supported on SQL Server, SQL Database, and SQL Data Warehouse. |
| PolyBase scale-out groups | Scale out parallelism between SQL Server and Hadoop by using SQL Server scale-out groups. |
| PolyBase installation | Reference and steps for installing PolyBase with the installation wizard or with a command-line tool. |
| PolyBase configuration | Configure SQL Server settings for PolyBase. For example, configure computation pushdown and kerberos security. |
| PolyBase T-SQL objects | Create the T-SQL objects that PolyBase uses to define and access external data. |
| PolyBase Queries | Use T-SQL statements to query, import, or export external data. |
| PolyBase troubleshooting | Techniques to manage PolyBase queries. Use dynamic management views (DMVs) to monitor PolyBase queries, and learn to read a PolyBase query plan to find performance bottlenecks. |

# SQL Server Replication

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Replication is a set of technologies for copying and distributing data and database objects from one database to another and then synchronizing between databases to maintain consistency. Use replication to distribute data to different locations and to remote or mobile users over local and wide area networks, dial-up connections, wireless connections, and the Internet.

Transactional replication is typically used in server-to-server scenarios that require high throughput, including: improving scalability and availability; data warehousing and reporting; integrating data from multiple sites; integrating heterogeneous data; and offloading batch processing. Merge replication is primarily designed for mobile applications or distributed server applications that have possible data conflicts. Common scenarios include: exchanging data with mobile users; consumer point of sale (POS) applications; and integration of data from multiple sites. Snapshot replication is used to provide the initial data set for transactional and merge replication; it can also be used when complete refreshes of data are appropriate. With these three types of replication, SQL Server provides a powerful and flexible system for synchronizing data across your enterprise. Replication to SQLCE 3.5 and SQLCE 4.0 is supported on both Windows Server 2012 and Windows 8.

As an alternative to replication, you can synchronize databases by using Microsoft Sync Framework. Sync Framework includes components and an intuitive and flexible API that make it easy to synchronize among SQL Server, SQL Server Express, SQL Server Compact, and SQL Azure databases. Sync Framework also includes classes that can be adapted to synchronize between a SQL Server database and any other database that is compatible with ADO.NET. For detailed documentation of the Sync Framework database synchronization components, see Synchronizing Databases. For an overview of Sync Framework, see Microsoft Sync Framework Developer Center. For a comparison between Sync Framework and Merge Replication, see Synchronizing Databases Overview

**Browse by area**

- What's New
- Backward Compatibility
- Replication Features and Tasks
- Technical Reference

# Resource Governor

5/3/2018 • 4 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

SQL Server Resource Governor is a feature than you can use to manage SQL Server workload and system resource consumption. Resource Governor enables you to specify limits on the amount of CPU, physical IO, and memory that incoming application requests can use.

## Benefits of Resource Governor

Resource Governor enables you to manage SQL Server workloads and resources by specifying limits on resource consumption by incoming requests. In the Resource Governor context, workload is a set of similarly sized queries or requests that can, and should be, treated as a single entity. This is not a requirement, but the more uniform the resource usage pattern of a workload is, the more benefit you are likely to derive from Resource Governor. Resource limits can be reconfigured in real time with minimal impact on workloads that are executing.

In an environment where multiple distinct workloads are present on the same server, Resource Governor enables you to differentiate these workloads and allocate shared resources as they are requested, based on the limits that you specify. These resources are CPU, physical IO, and memory.

By using Resource Governor, you can:

- Provide multitenancy and resource isolation on single instances of SQL Server that serve multiple client workloads. That is, you can divide the available resources on a server among the workloads and minimize the problems that can occur when workloads compete for resources.
- Provide predictable performance and support SLAs for workload tenants in a multi-workload and multi-user environment.
- Isolate and limit runaway queries or throttle IO resources for operations such as DBCC CHECKDB that can saturate the IO subsystem and negatively impact other workloads.
- Add fine-grained resource tracking for resource usage chargebacks and provide predictable billing to the consumers of the server resources.

## Resource Governor Constraints

This release of Resource Governor has the following constraints:

- Resource management is limited to the SQL Server Database Engine. Resource Governor can not be used for Analysis Services, Integration Services, and Reporting Services.
- There is no workload monitoring or workload management between SQL Server instances.
- Resource Governor can manage OLTP workloads but these types of queries, which are typically very short in duration, are not always on the CPU long enough to apply bandwidth controls. This may skew in the statistics returned for CPU usage %.
- The ability to govern physical IO only applies to user operations and not system tasks. System tasks include write operations to the transaction log and Lazy Writer IO operations. The Resource Govenor applies primarily to user read operations because most write operations are typically performed by system tasks.

- You cannot set IO thresholds on the internal resource pool.

## Resource Concepts

The following three concepts are fundamental to understanding and using Resource Governor:

- **Resource pools.** A resource pool, represents the physical resources of the server. You can think of a pool as a virtual SQL Server instance inside of a SQL Server instance. Two resource pools (internal and default) are created when SQL Server is installed. Resource Governor also supports user-defined resource pools. For more information, see Resource Governor Resource Pool.

- **Workload groups.** A workload group serves as a container for session requests that have similar classification criteria. A workload allows for aggregate monitoring of the sessions, and defines policies for the sessions. Each workload group is in a resource pool. Two workload groups (internal and default) are created and mapped to their corresponding resource pools when SQL Server is installed. Resource Governor also supports user-defined workload groups. For more information see, Resource Governor Workload Group.

- **Classification.** The Classification process assigns incoming sessions to a workload group based on the characteristics of the session. You can tailor the classification logic by writing a user-defined function, called a classifier function. Resource Governor also supports a classifier user-defined function for implementing classification rules. For more information, see Resource Governor Classifier Function.

> **NOTE**
>
> Resource Governor does not impose any controls on a dedicated administrator connection (DAC). There is no need to classify DAC queries, which run in the internal workload group and resource pool.

In the context of Resource Governor, you can treat the preceding concepts as components. The following illustration shows these components and their relationship with each other as they exist in the database engine environment. From a processing perspective, the simplified flow is as follows:

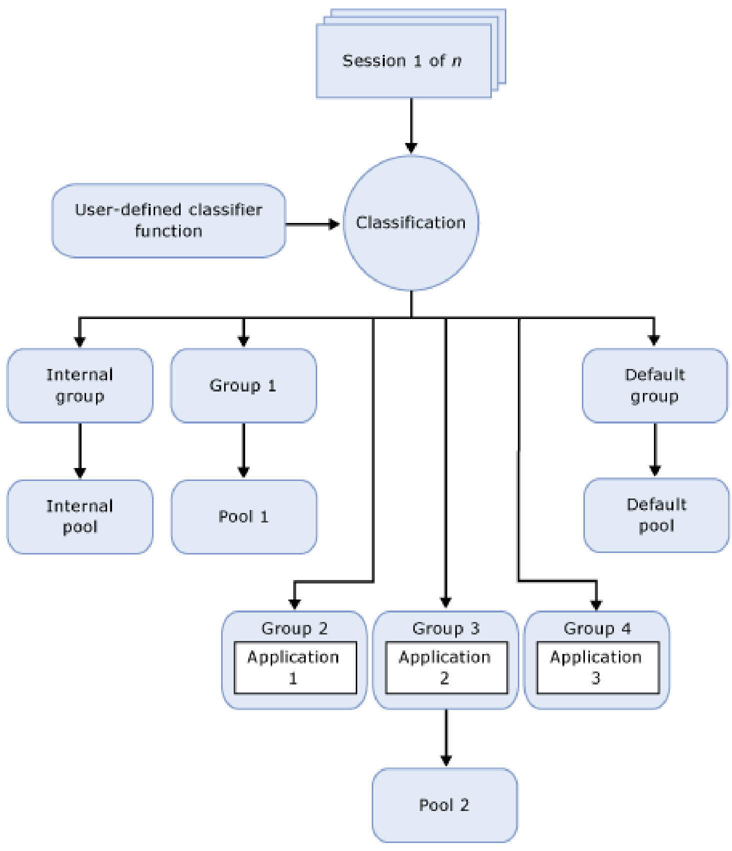
- There is an incoming connection for a session (Session 1 of *n*).

- The session is classified (Classification).

- The session workload is routed to a workload group, for example, Group 4.

- The workload group uses the resource pool it is associated with, for example, Pool 2.

- The resource pool provides and limits the resources required by the application, for example, Application 3.




## Resource Governor Tasks

| TASK DESCRIPTION | TOPIC |
|---|---|
| Describes how to enable Resource Governor. | Enable Resource Governor |
| Describes how to disable Resource Governor. | Disable Resource Governor |
| Describes how to create, alter, and drop a resource pool. | Resource Governor Resource Pool |
| Describes how to create, alter, move, and drop a workload group. | Resource Governor Workload Group |
| Describes how to create and test a classifier user-defined function. | Resource Governor Classifier Function |
| Describes how to configure Resource Governor using a template. | Configure Resource Governor Using a Template |
| Describes how to view Resource Governor properties. | View Resource Governor Properties |

## See Also

Database Engine Instances (SQL Server)

# Database Engine Scripting

5/3/2018 • 3 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

The SQL Server Database Engine supports the Microsoft PowerShell scripting environment to manage instances of the Database Engine and the objects in the instances. You can also build and run Database Engine queries that contain Transact-SQL and XQuery in environments very similar to scripting environments.

## SQL Server PowerShell

SQL Server includes two SQL Server PowerShell snap-ins that implement:

- A SQL Server PowerShell provider that exposes the SQL Server management object model hierarchies as PowerShell paths that are similar to file system paths. You can use the SQL Server management object model classes to manage the objects represented at each node of the path.
- A set of SQL Server cmdlets that implement SQL Server commands. One of the cmdlets is **Invoke-Sqlcmd**. This is used to run Database Engine Query scripts to be run with the **sqlcmd** utility.

  SQL Server provides these features for running PowerShell:

- The **sqlps** PowerShell module that can be imported to a PowerShell session, the module then loads the SQL Server snap-ins. You can interactively run ad hoc PowerShell commands. You can run script files using a command such as .\MyFolder\MyScript.ps1.
- PowerShell script files can be used as input to SQL Server Agent PowerShell job steps that run the scripts either at scheduled intervals or in response to system events.
- The **sqlps** utility that starts PowerShell and imports the SQL Server module. You can then perform all actions supported by the module. You can start the **sqlps** utility either in a command prompt or by right-clicking on the nodes in the SQL Server Management Studio Object Explorer tree and selecting **Start PowerShell**.

## Database Engine Queries

Database Engine query scripts contain three types of elements:

- Transact-SQL language statements.
- XQuery language statements
- Commands and variables from the **sqlcmd** utility.

  SQL Server provides three environments for building and running Database Engine queries:

- You can interactively run and debug Database Engine queries in the Database Engine Query Editor in SQL Server Management Studio. You can code and debug several statements in one session, then save all of the statements in a single script file.
- The **sqlcmd** command prompt utility lets you interactively run Database Engine queries, and also run existing Database Engine query script files.

  Database Engine query script files are typically coded interactively in SQL Server Management Studio by

using the Database Engine Query Editor. The file can later be opened in one of these environments:

- Use the SQL Server Management Studio **File/Open** menu to open the file in a new Database Engine Query Editor window.
- Use the **-i***input_file* parameter to run the file with the **sqlcmd** utility.
- Use the **-QueryFromFile** parameter to run the file with the **Invoke-Sqlcmd** cmdlet in SQL Server PowerShell scripts.
- Use SQL Server Agent Transact-SQL job steps to run the scripts either at scheduled intervals or in response to system events.

In addition, you can use the SQL Server Generate Script Wizard to generate Transact-SQL scripts. You can right-click objects in the SQL Server Management Studio Object Explorer, then select the **Generate Script** menu item. **Generate Script** launches the wizard, which guides you through the process of creating a script.

## Database Engine Scripting Tasks

| TASK DESCRIPTION | TOPIC |
| --- | --- |
| Describes how to use the code and text editors in Management Studio to interactively develop, debug, and run Transact-SQL scripts | Query and Text Editors (SQL Server Management Studio) |
| Describes how to use the **sqlcmd** utility to run Transact-SQL scripts from the command prompt, including the ability to interactively develop scripts. | sqlcmd How-to Topics |
| Describes how to integrate the SQL Server components into a Windows PowerShell environment and then build PowerShell scripts for managing SQL Server instances and objects. | SQL Server PowerShell |
| Describes how to use the **Generate and Publish Scripts** wizard to create Transact-SQL scripts that recreate one or more of the objects from a database. | Generate Scripts (SQL Server Management Studio) |

## See Also

sqlcmd Utility
Tutorial: Writing Transact-SQL Statements

# Full-Text Search

5/3/2018 • 16 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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.

## Basic tasks

This topic provides an overview of Full-Text Search and describes its components and its architecture. If you prefer to get started right away, here are the basic tasks.

- Get Started with Full-Text Search
- Create and Manage Full-Text Catalogs
- Create and Manage Full-Text Indexes
- Populate Full-Text Indexes
- Query with Full-Text Search

> **NOTE**
>
> Full-Text Search is an optional component of the SQL Server Database Engine. If you didn't select Full-Text Search when you installed SQL Server, run SQL Server Setup again to add it.

## Overview

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;
```

  For more information, see Query with Full-Text Search.

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

In contrast to full-text search, the LIKE 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.

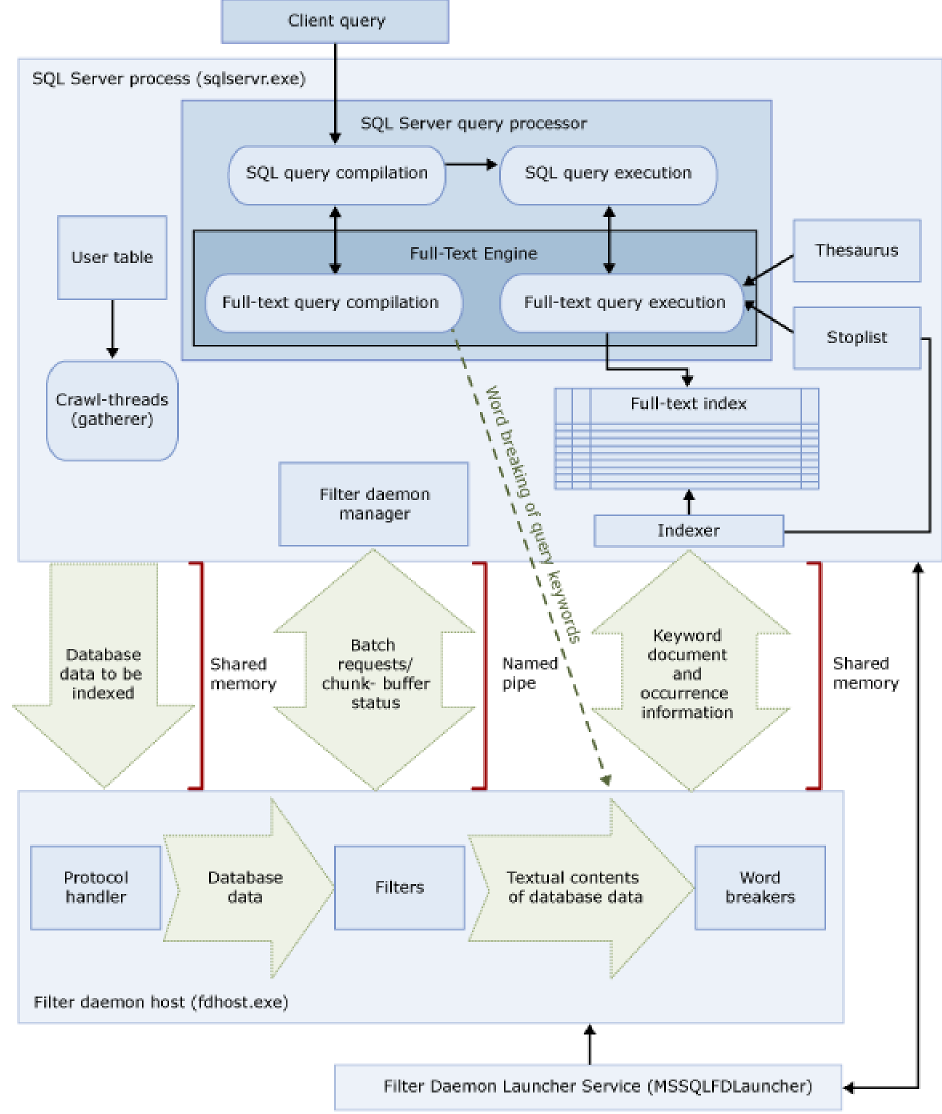
## Full-Text Search architecture

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. For information about setting the service account for this service, see Set the Service Account for the Full-text Filter Daemon Launcher.

  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. For more information, see Configure and Manage Thesaurus Files for Full-Text Search.
- **Stoplist objects.** Stoplist objects contain a list of common words that are not useful for the search. For more information, see Configure and Manage Stopwords and Stoplists for Full-Text 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.

> **NOTE**
>
> In SQL Server 2008 and later versions, the Full-Text Engine resides in the SQL Server process, rather than in a separate service. Integrating the Full-Text Engine into the Database Engine improved full-text manageability, optimization of mixed query, and overall performance.

- **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.
- **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. For more information, see Configure and Manage Word Breakers and Stemmers for Search.

## 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; for more information, see Upgrade Full-Text Search.

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.

#### Full-text index structure

A good understanding of the structure of a full-text index will help you understand how the Full-Text Engine works. This topic uses the following excerpt of the **Document** table in Adventure Works as an example table. This excerpt shows only two columns, the **DocumentID** column and the **Title** column, and three rows from the table.

For this example, we will assume that a full-text index has been created on the **Title** column.

| DOCUMENTID | TITLE |
| --- | --- |
| 1 | Crank Arm and Tire Maintenance |

| DOCUMENTID | TITLE |
|---|---|
| 2 | Front Reflector Bracket and Reflector Assembly 3 |
| 3 | Front Reflector Bracket Installation |

For example, the following table, which shows Fragment 1, depicts the contents of the full-text index created on the **Title** column of the **Document** table. Full-text indexes contain more information than is presented in this table. The table is a logical representation of a full-text index and is provided for demonstration purposes only. The rows are stored in a compressed format to optimize disk usage.

Notice that the data has been inverted from the original documents. Inversion occurs because the keywords are mapped to the document IDs. For this reason, a full-text index is often referred to as an inverted index.

Also notice that the keyword "and" has been removed from the full-text index. This is done because "and" is a stopword, and removing stopwords from a full-text index can lead to substantial savings in disk space thereby improving query performance. For more information about stopwords, see Configure and Manage Stopwords and Stoplists for Full-Text Search.

**Fragment 1**

| KEYWORD | COLID | DOCID | OCCURRENCE |
|---|---|---|---|
| Crank | 1 | 1 | 1 |
| Arm | 1 | 1 | 2 |
| Tire | 1 | 1 | 4 |
| Maintenance | 1 | 1 | 5 |
| Front | 1 | 2 | 1 |
| Front | 1 | 3 | 1 |
| Reflector | 1 | 2 | 2 |
| Reflector | 1 | 2 | 5 |
| Reflector | 1 | 3 | 2 |
| Bracket | 1 | 2 | 3 |
| Bracket | 1 | 3 | 3 |
| Assembly | 1 | 2 | 6 |
| 3 | 1 | 2 | 7 |
| Installation | 1 | 3 | 4 |

The **Keyword** column contains a representation of a single token extracted at indexing time. Word breakers determine what makes up a token.

The **ColId** column contains a value that corresponds to a particular column that is full-text indexed.

The **DocId** column contains values for an eight-byte integer that maps to a particular full-text key value in a full-text indexed table. This mapping is necessary when the full-text key is not an integer data type. In such cases, mappings between full-text key values and **DocId** values are maintained in a separate table called the DocId Mapping table. To query for these mappings use the sp_fulltext_keymappings system stored procedure. To satisfy a search condition, DocId values from the above table need to be joined with the DocId Mapping table to retrieve rows from the base table being queried. If the full-text key value of the base table is an integer type, the value directly serves as the DocId and no mapping is necessary. Therefore, using integer full-text key values can help optimize full-text queries.

The **Occurrence** column contains an integer value. For each DocId value, there is a list of occurrence values that correspond to the relative word offsets of the particular keyword within that DocId. Occurrence values are useful in determining phrase or proximity matches, for example, phrases have numerically adjacent occurrence values. They are also useful in computing relevance scores; for example, the number of occurrences of a keyword in a DocId may be used in scoring.

### Full-text index fragments

The logical full-text index is usually split across multiple internal tables. Each internal table is called a full-text index fragment. Some of these fragments might contain newer data than others. For example, if a user updates the following row whose DocId is 3 and the table is auto change-tracked, a new fragment is created.

| DOCUMENTID | TITLE |
| --- | --- |
| 3 | Rear Reflector |

In the following example, which shows Fragment 2, the fragment contains newer data about DocId 3 compared to Fragment 1. Therefore, when the user queries for "Rear Reflector" the data from Fragment 2 is used for DocId 3. Each fragment is marked with a creation timestamp that can be queried by using the sys.fulltext_index_fragments catalog view.

**Fragment 2**

| KEYWORD | COLID | DOCID | OCC |
| --- | --- | --- | --- |
| Rear | 1 | 3 | 1 |
| Reflector | 1 | 3 | 2 |

As can be seen from Fragment 2, full-text queries need to query each fragment internally and discard older entries. Therefore, too many full-text index fragments in the full-text index can lead to substantial degradation in query performance. To reduce the number of fragments, reorganize the fulltext catalog by using the REORGANIZE option of the ALTER FULLTEXT CATALOG Transact-SQL statement. This statement performs a *master merge*, which merges the fragments into a single larger fragment and removes all obsolete entries from the full-text index.

After being reorganized, the example index would contain the following rows:

| KEYWORD | COLID | DOCID | OCC |
| --- | --- | --- | --- |
| Crank | 1 | 1 | 1 |
| Arm | 1 | 1 | 2 |
| Tire | 1 | 1 | 4 |

| KEYWORD | COLID | DOCID | OCC |
| --- | --- | --- | --- |
| Maintenance | 1 | 1 | 5 |
| Front | 1 | 2 | 1 |
| Rear | 1 | 3 | 1 |
| Reflector | 1 | 2 | 2 |
| Reflector | 1 | 2 | 5 |
| Reflector | 1 | 3 | 2 |
| Bracket | 1 | 2 | 3 |
| Assembly | 1 | 2 | 6 |
| 3 | 1 | 2 | 7 |

**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). 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.
- **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.

- **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.

- **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.

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

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2008) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

# Security Center for SQL Server Database Engine and Azure SQL Database

5/3/2018 • 3 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

This page provides links to help you locate the information that you need about security and protection in the SQL Server Database Engine and Azure SQL Database.

### Legend

indicates that the feature is available only in SQL Server Database Engine (both SQL Server running on-premises and SQL Server running in an Azure Virtual Machine).

indicates that the feature is available only in Azure SQL Database.

indicates that the feature is available in both SQL Server and SQL Database.

## Authentication: Who are you?

| | |
|---|---|
| **Who Authenticates?**<br><br>Windows Authentication<br><br>SQL Server Authentication<br><br>Azure Active Directory | Who Authenticates? (Windows or SQL Server)<br><br>Choose an Authentication Mode<br><br>Connecting to SQL Database By Using Azure Active Directory Authentication |
| **Where Authenticated?**<br><br>At master Database: Logins and DB Users<br><br>At User Database: Contained DB Users | Authenticate at the master database (Logins and database users)<br><br>Create a SQL Server Login<br><br>Managing Databases and Logins in Azure SQL Database<br><br>Create a Database User<br><br>Authenticate at a user database<br><br>Contained Database Users - Making Your Database Portable |
| **Using Other Identities**<br><br>Credentials<br><br>Execute as Another Login<br><br>Execute as Another Database User | Credentials (Database Engine)<br><br>Execute as Another Login<br><br>Execute as Another Database User |

## Authorization: What can you do?

| | |
|---|---|
| **Granting, Revoking, and Denying Permissions**<br>Securable Classes<br>Granular Server Permissions<br>Granular Database Permissions | Permissions Hierarchy (Database Engine)<br>Permissions<br>Securables<br>Getting Started with Database Engine Permissions |
| **Security by Roles**<br>Server Level Roles<br>Database Level Roles | Server-Level Roles<br>Database-Level Roles |
| **Restricting Data Access to Selected Data Elements**<br>Restrict Data Access With Views/Procedures<br>Row-Level Security<br>Dynamic Data Masking<br>Signed Objects | Restrict Data Access Using Views and Procedures<br>Row-Level Security (SQL Server)<br>Row-Level Security (Azure SQL Database)<br>Dynamic Data Masking (SQL Server)<br>Dynamic Data Masking (Azure SQL Database)<br>Signed Objects |

## Encryption: Storing Secret Data

| | |
|---|---|
| **Encrypting Files**<br>BitLocker Encryption (Drive Level)<br>NTFS Encryption (Folder Level)<br>Transparent Data Encryption (File Level)<br>Backup Encryption (File Level) | BitLocker (Drive Level)<br>NTFS Encryption (Folder Level)<br>Transparent Data Encryption (File Level)<br>Backup Encryption (File Level) |
| **Encrypting Sources**<br>Extensible Key Management Module<br>Keys Stored in the Azure Key Vault<br>Always Encrypted | Extensible Key Management Module<br>Keys Stored in the Azure Key Vault<br>Always Encrypted |

| | |
|---|---|
| **Column, Data, & Key Encryption**<br><br>Encrypt by Certificate<br><br>Encrypt by Symmetric Key<br><br>Encrypt by Asymmetric Key<br><br>Encrypt by Passphrase | Encrypt by Certificate<br><br>Encrypt by Asymmetric Key<br><br>Encrypt by Symmetric Key<br><br>Encrypt by Passphrase<br><br>Encrypt a Column of Data |

## Connection Security: Restricting and Securing

| | |
|---|---|
| **Firewall Protection**<br><br>Windows Firewall Settings<br><br>Azure Service Firewall Settings<br><br>Database Firewall Settings | Configure a Windows Firewall for Database Engine Access<br><br>Azure SQL Database Firewall Settings<br><br>Azure Service Firewall Settings |
| **Encrypting Data in Transit**<br><br>Forced SSL Connections<br><br>Optional SSL Connections | Secure Sockets Layer for the Database Engine<br><br>Secure Sockets Layer for SQL Database<br><br>TLS 1.2 support for Microsoft SQL Server |

## Auditing: Recording Access

| | |
|---|---|
| **Automated Auditing**<br><br>SQL Server Audit (Server and DB Level)<br><br>SQL Database Audit (Database Level)<br><br>Threat Detection | SQL Server Audit (Database Engine)<br><br>SQL Database Auditing<br><br>Get started with SQL Database Threat Detection<br><br>SQL Database Vulnerability Assessment |
| **Custom Audit**<br><br>Triggers | Custom Audit Implementation: Creating DDL Triggers and DML Triggers |
| **Compliance**<br><br>Compliance | SQL Server:<br>Common Criteria<br><br>SQL Database:<br>Microsoft Azure Trust Center: Compliance by Feature |

## SQL Injection

SQL injection is an attack in which malicious code is inserted into strings that are later passed to the Database

Engine for parsing and execution. Any procedure that constructs SQL statements should be reviewed for injection vulnerabilities because SQL Server will execute all syntactically valid queries that it receives. All database systems have some risk of SQL Injection, and many of the vulnerabilities are introduced in the application that is querying the Database Engine. You can thwart SQL injection attacks by using stored procedures and parameterized commands, avoiding dynamic SQL, and restricting permissions on all users. For more information, see SQL Injection.

Additional links for application programmers:

- Application Security Scenarios in SQL Server
- Writing Secure Dynamic SQL in SQL Server
- How To: Protect From SQL Injection in ASP.NET

## See Also

Getting Started with Database Engine Permissions
Securing SQL Server
Principals (Database Engine)
SQL Server Certificates and Asymmetric Keys
SQL Server Encryption
Surface Area Configuration
Strong Passwords
TRUSTWORTHY Database Property
Database Engine Features and Tasks
Protecting Your SQL Server Intellectual Property

# Sequence Numbers

5/3/2018 • 10 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

A sequence is a user-defined schema-bound object that generates a sequence of numeric values according to the specification with which the sequence was created. 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. User applications can reference a sequence object and coordinate the values keys across multiple rows and tables.

A sequence is created independently of the tables by using the **CREATE SEQUENCE** statement. Options enable you to control the increment, maximum and minimum values, starting point, automatic restarting capability, and caching to improve performance. For information about the options, see CREATE SEQUENCE.

Unlike identity column values, which are generated when rows are inserted, an application can obtain the next sequence number before inserting the row by calling the NEXT VALUE FOR function. The sequence number is allocated when NEXT VALUE FOR is called even if the number is never inserted into a table. The NEXT VALUE FOR function can be used as the default value for a column in a table definition. Use sp_sequence_get_range to get a range of multiple sequence numbers at once.

A sequence can be defined as any integer data type. If the data type is not specified, a sequence defaults to **bigint**.

## 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.

Uniqueness is not automatically enforced for sequence values. The ability to reuse sequence values is by design. If sequence values in a table are required to be unique, create a unique index on the column. If sequence values in a table are required to be unique throughout a group of tables, create triggers to prevent duplicates caused by update statements or sequence number cycling.

The sequence object generates numbers according to its definition, but the sequence object does not control how the numbers are used. Sequence numbers inserted into a table can have gaps when a transaction is rolled back, when a sequence object is shared by multiple tables, or when sequence numbers are allocated without using them in tables. When created with the CACHE option, an unexpected shutdown, such as a power failure, can lose the sequence numbers in the cache.

If there are multiple instances of the **NEXT VALUE FOR** function specifying the same sequence generator within a single Transact-SQL statement, all those instances return the same value for a given row processed by that Transact-SQL statement. This behavior is consistent with the ANSI standard.

## Typical Use

To create an integer sequence number that increments by 1 from -2,147,483,648 to 2,147,483,647, use the following statement.

```
CREATE SEQUENCE Schema.SequenceName
    AS int
    INCREMENT BY 1 ;
```

To create an integer sequence number similar to an identity column that increments by 1 from 1 to 2,147,483,647, use the following statement.

```
CREATE SEQUENCE Schema.SequenceName
    AS int
    START WITH 1
    INCREMENT BY 1 ;
```

## Managing Sequences

For information about sequences, query sys.sequences.

## Examples

There are additional examples in the topics CREATE SEQUENCE (Transact-SQL), NEXT VALUE FOR (Transact-SQL), and sp_sequence_get_range.

### A. Using a sequence number in a single table

The following example creates a schema named Test, a table named Orders, and a sequence named CountBy1, and then inserts rows into the table using the NEXT VALUE FOR function.

```
--Create the Test schema
CREATE SCHEMA Test ;
GO

-- Create a table
CREATE TABLE Test.Orders
    (OrderID int PRIMARY KEY,
    Name varchar(20) NOT NULL,
    Qty int NOT NULL);
GO

-- Create a sequence
CREATE SEQUENCE Test.CountBy1
    START WITH 1
    INCREMENT BY 1 ;
GO

-- Insert three records
INSERT Test.Orders (OrderID, Name, Qty)
    VALUES (NEXT VALUE FOR Test.CountBy1, 'Tire', 2) ;
INSERT test.Orders (OrderID, Name, Qty)
    VALUES (NEXT VALUE FOR Test.CountBy1, 'Seat', 1) ;
INSERT test.Orders (OrderID, Name, Qty)
    VALUES (NEXT VALUE FOR Test.CountBy1, 'Brake', 1) ;
GO

-- View the table
SELECT * FROM Test.Orders ;
GO
```

Here is the result set.

`OrderID Name Qty`

`1 Tire 2`

`2 Seat 1`

`3 Brake 1`

### B. Calling NEXT VALUE FOR before inserting a row

Using the `Orders` table created in example A, the following example declares a variable named `@nextID`, and then uses the NEXT VALUE FOR function to set the variable to the next available sequence number. The application is presumed to do some processing of the order, such as providing the customer with the `OrderID` number of their potential order, and then validates the order. No matter how long this processing might take, or how many other orders are added during the process, the original number is preserved for use by this connection. Finally, the `INSERT` statement adds the order to the `Orders` table.

```
DECLARE @NextID int ;
SET @NextID = NEXT VALUE FOR Test.CountBy1;
-- Some work happens
INSERT Test.Orders (OrderID, Name, Qty)
    VALUES (@NextID, 'Rim', 2) ;
GO
```

### C. Using a sequence number in multiple tables

This example assumes that a production-line monitoring process receives notification of events that occur throughout the workshop. Each event receives a unique and monotonically increasing `EventID` number. All events use the same `EventID` sequence number so that reports that combine all events can uniquely identify each event. However the event data is stored in three different tables, depending on the type of event. The code example

creates a schema named `Audit`, a sequence named `EventCounter`, and three tables which each use the `EventCounter` sequence as a default value. Then the example adds rows to the three tables and queries the results.

```
CREATE SCHEMA Audit ;
GO
CREATE SEQUENCE Audit.EventCounter
    AS int
    START WITH 1
    INCREMENT BY 1 ;
GO

CREATE TABLE Audit.ProcessEvents
(
    EventID int PRIMARY KEY CLUSTERED
        DEFAULT (NEXT VALUE FOR Audit.EventCounter),
    EventTime datetime NOT NULL DEFAULT (getdate()),
    EventCode nvarchar(5) NOT NULL,
    Description nvarchar(300) NULL
) ;
GO

CREATE TABLE Audit.ErrorEvents
(
    EventID int PRIMARY KEY CLUSTERED
        DEFAULT (NEXT VALUE FOR Audit.EventCounter),
    EventTime datetime NOT NULL DEFAULT (getdate()),
    EquipmentID int NULL,
    ErrorNumber int NOT NULL,
    EventDesc nvarchar(256) NULL
) ;
GO

CREATE TABLE Audit.StartStopEvents
(
    EventID int PRIMARY KEY CLUSTERED
        DEFAULT (NEXT VALUE FOR Audit.EventCounter),
    EventTime datetime NOT NULL DEFAULT (getdate()),
    EquipmentID int NOT NULL,
    StartOrStop bit NOT NULL
) ;
GO

INSERT Audit.StartStopEvents (EquipmentID, StartOrStop)
    VALUES (248, 0) ;
INSERT Audit.StartStopEvents (EquipmentID, StartOrStop)
    VALUES (72, 0) ;
INSERT Audit.ProcessEvents (EventCode, Description)
    VALUES (2735,
    'Clean room temperature 18 degrees C.') ;
INSERT Audit.ProcessEvents (EventCode, Description)
    VALUES (18, 'Spin rate threashold exceeded.') ;
INSERT Audit.ErrorEvents (EquipmentID, ErrorNumber, EventDesc)
    VALUES (248, 82, 'Feeder jam') ;
INSERT Audit.StartStopEvents (EquipmentID, StartOrStop)
    VALUES (248, 1) ;
INSERT Audit.ProcessEvents (EventCode, Description)
    VALUES (1841, 'Central feed in bypass mode.') ;
-- The following statement combines all events, though not all fields.
SELECT EventID, EventTime, Description FROM Audit.ProcessEvents
UNION SELECT EventID, EventTime, EventDesc FROM Audit.ErrorEvents
UNION SELECT EventID, EventTime,
CASE StartOrStop
    WHEN 0 THEN 'Start'
    ELSE 'Stop'
END
FROM Audit.StartStopEvents
ORDER BY EventID ;
GO
```

Here is the result set.

`EventID EventTime Description`

`1 2009-11-02 15:00:51.157 Start`

`2 2009-11-02 15:00:51.160 Start`

`3 2009-11-02 15:00:51.167 Clean room temperature 18 degrees C.`

`4 2009-11-02 15:00:51.167 Spin rate threshold exceeded.`

`5 2009-11-02 15:00:51.173 Feeder jam`

`6 2009-11-02 15:00:51.177 Stop`

`7 2009-11-02 15:00:51.180 Central feed in bypass mode.`

### D. Generating repeating sequence numbers in a result set

The following example demonstrates two features of sequence numbers: cycling, and using `NEXT VALUE FOR` in a select statement.

```
CREATE SEQUENCE CountBy5
   AS tinyint
    START WITH 1
    INCREMENT BY 1
    MINVALUE 1
    MAXVALUE 5
    CYCLE ;
GO

SELECT NEXT VALUE FOR CountBy5 AS SurveyGroup, Name FROM sys.objects ;
GO
```

### E. Generating sequence numbers for a result set by using the OVER clause

The following example uses the `OVER` clause to sort the result set by `Name` before it adds the sequence number column.

```
USE AdventureWorks2012 ;
GO

CREATE SCHEMA Samples ;
GO

CREATE SEQUENCE Samples.IDLabel
    AS tinyint
    START WITH 1
    INCREMENT BY 1 ;
GO

SELECT NEXT VALUE FOR Samples.IDLabel OVER (ORDER BY Name) AS NutID, ProductID, Name, ProductNumber FROM
Production.Product
WHERE Name LIKE '%nut%' ;
```

### F. Resetting the sequence number

Example E consumed the first 79 of the `Samples.IDLabel` sequence numbers. (Your version of `AdventureWorks2012` may return a different number of results.) Execute the following to consume the next 79 sequence numbers (80 though 158).

```
SELECT NEXT VALUE FOR Samples.IDLabel OVER (ORDER BY Name) AS NutID, ProductID, Name, ProductNumber FROM
Production.Product
WHERE Name LIKE '%nut%' ;
```

Execute the following statement to restart the `Samples.IDLabel` sequence.

```
ALTER SEQUENCE Samples.IDLabel
RESTART WITH 1 ;
```

Execute the select statement again to verify that the `Samples.IDLabel` sequence restarted with number 1.

```
SELECT NEXT VALUE FOR Samples.IDLabel OVER (ORDER BY Name) AS NutID, ProductID, Name, ProductNumber FROM
Production.Product
WHERE Name LIKE '%nut%' ;
```

### G. Changing a table from identity to sequence

The following example creates a schema and table containing three rows for the example. Then the example adds a new column and drops the old column.

```
-- Create a schema
CREATE SCHEMA Test ;
GO

-- Create a table
CREATE TABLE Test.Department
    (
        DepartmentID smallint IDENTITY(1,1) NOT NULL,
        Name nvarchar(100) NOT NULL,
        GroupName nvarchar(100) NOT NULL
    CONSTRAINT PK_Department_DepartmentID PRIMARY KEY CLUSTERED
         (DepartmentID ASC)
    ) ;
GO

-- Insert three rows into the table
INSERT Test.Department(Name, GroupName)
    VALUES ('Engineering', 'Research and Development');
GO

INSERT Test.Department(Name, GroupName)
    VALUES ('Tool Design', 'Research and Development');
GO

INSERT Test.Department(Name, GroupName)
    VALUES ('Sales', 'Sales and Marketing');
GO

-- View the table that will be changed
SELECT * FROM Test.Department ;
GO

-- End of portion creating a sample table
--------------------------------------------------------
-- Add the new column that does not have the IDENTITY property
ALTER TABLE Test.Department
    ADD DepartmentIDNew smallint NULL
GO

-- Copy values from the old column to the new column
UPDATE Test.Department
    SET DepartmentIDNew = DepartmentID ;
```

```
GO

-- Drop the primary key constraint on the old column
ALTER TABLE Test.Department
    DROP CONSTRAINT [PK_Department_DepartmentID];
-- Drop the old column
ALTER TABLE Test.Department
    DROP COLUMN DepartmentID ;
GO

-- Rename the new column to the old columns name
EXEC sp_rename 'Test.Department.DepartmentIDNew',
    'DepartmentID', 'COLUMN';
GO

-- Change the new column to NOT NULL
ALTER TABLE Test.Department
    ALTER COLUMN DepartmentID smallint NOT NULL ;
-- Add the unique primary key constraint
ALTER TABLE Test.Department
    ADD CONSTRAINT PK_Department_DepartmentID PRIMARY KEY CLUSTERED
         (DepartmentID ASC) ;
-- Get the highest current value from the DepartmentID column
-- and create a sequence to use with the column. (Returns 3.)
SELECT MAX(DepartmentID) FROM Test.Department ;
-- Use the next desired value (4) as the START WITH VALUE;
CREATE SEQUENCE Test.DeptSeq
    AS smallint
    START WITH 4
    INCREMENT BY 1 ;
GO

-- Add a default value for the DepartmentID column
ALTER TABLE Test.Department
    ADD CONSTRAINT DefSequence DEFAULT (NEXT VALUE FOR Test.DeptSeq)
        FOR DepartmentID;
GO

-- View the result
SELECT DepartmentID, Name, GroupName
FROM Test.Department ;
-- Test insert
INSERT Test.Department (Name, GroupName)
    VALUES ('Audit', 'Quality Assurance') ;
GO

-- View the result
SELECT DepartmentID, Name, GroupName
FROM Test.Department ;
GO
```

Transact-SQL statements that use `SELECT *` will receive the new column as the last column instead of the first column. If this is not acceptable, then you must create an entirely new table, move the data to it, and then recreate the permissions on the new table.

# Related Content

CREATE SEQUENCE (Transact-SQL)

ALTER SEQUENCE (Transact-SQL)

DROP SEQUENCE (Transact-SQL)

IDENTITY (Property) (Transact-SQL)

# Event Notifications

5/3/2018 • 5 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Event notifications send information about events to a Service Broker service. Event notifications execute in response to a variety of Transact-SQL data definition language (DDL) statements and SQL Trace events by sending information about these events to a Service Broker service.

Event notifications can be used to do the following:

- Log and review changes or activity occurring on the database.
- Perform an action in response to an event in an asynchronous instead of synchronous manner.

  Event notifications can offer a programming alternative to DDL triggers and SQL Trace.

## Event Notifications Benefits

Event notifications run asynchronously, outside the scope of a transaction. Therefore, unlike DDL triggers, event notifications can be used inside a database application to respond to events without using any resources defined by the immediate transaction.

Unlike SQL Trace, event notifications can be used to perform an action inside an instance of SQL Server in response to a SQL Trace event.

Event data can be used by applications that are running together with SQL Server to track progress and make decisions. For example, the following event notification sends a notice to a certain service every time an `ALTER TABLE` statement is issued in the **AdventureWorks2012** sample database.

```
USE AdventureWorks2012;
GO
CREATE EVENT NOTIFICATION NotifyALTER_T1
ON DATABASE
FOR ALTER_TABLE
TO SERVICE '//Adventure-Works.com/ArchiveService' ,
    '8140a771-3c4b-4479-8ac0-81008ab17984';
```

## Event Notifications Concepts

When an event notification is created, one or more Service Broker conversations between an instance of SQL Server and the target service you specify are opened. The conversations typically remain open as long as the event notification exists as an object on the server instance. In some error cases the conversations can close before the event notification is dropped. These conversations are never shared between event notifications. Every event notification has its own exclusive conversations. Ending a conversation explicitly prevents the target service from receiving more messages, and the conversation will not reopen the next time the event notification fires.

Event information is delivered to the Service Broker service as a variable of type **xml** that provides information about when an event occurs, about the database object affected, the Transact-SQL batch statement involved, and other information. For more information about the XML schema produced by event notifications, see EVENTDATA (Transact-SQL).

**Event Notifications vs. Triggers**

The following table compares and contrasts triggers and event notifications.

| TRIGGERS | EVENT NOTIFICATIONS |
|---|---|
| DML triggers respond to data manipulation language (DML) events. DDL triggers respond to data definition language (DDL) events. | Event notifications respond to DDL events and a subset of SQL trace events. |
| Triggers can run Transact-SQL or common language runtime (CLR) managed code. | Event notifications do not run code. Instead, they send **xml** messages to a Service Broker service. |
| Triggers are processed synchronously, within the scope of the transactions that cause them to fire. | Event notifications may be processed asynchronously and do not run in the scope of the transactions that cause them to fire. |
| The consumer of a trigger is tightly coupled with the event that causes it to fire. | The consumer of an event notification is decoupled from the event that causes it to fire. |
| Triggers must be processed on the local server. | Event notifications can be processed on a remote server. |
| Triggers can be rolled back. | Event notifications cannot be rolled back. |
| DML trigger names are schema-scoped. DDL trigger names are database-scoped or server-scoped. | Event notification names are scoped by the server or database. Event notifications on a QUEUE_ACTIVATION event are scoped to a specific queue. |
| DML triggers are owned by the same owner as the tables on which they are applied. | The owner of an event notification on a queue may have a different owner than the object on which it is applied. |
| Triggers support the EXECUTE AS clause. | Event notifications do not support the EXECUTE AS clause. |
| DDL trigger event information can be captured using the EVENTDATA function, which returns an **xml** data type. | Event notifications send **xml** event information to a Service Broker service. The information is formatted to the same schema as that of the EVENTDATA function. |
| Metadata about triggers is found in the **sys.triggers** and **sys.server_triggers** catalog views. | Metadata about event notifications is found in the **sys.event_notifications** and **sys.server_event_notifications** catalog views. |

### Event Notifications vs. SQL Trace

The following table compares and contrasts using event notifications and SQL Trace for monitoring server events.

| SQL TRACE | EVENT NOTIFICATIONS |
|---|---|
| SQL Trace generates no performance overhead associated with transactions. Packaging of data is efficient. | There is performance overhead associated with creating the XML-formatted event data and sending the event notification. |
| SQL Trace can monitor any trace event class. | Event notifications can monitor a subset of trace event classes and also all data definition language (DDL) events. |
| You can customize which data columns to generate in a trace event. | The schema of the XML-formatted event data returned by event notifications is fixed. |
| Trace events generated by DDL are always generated, regardless of whether the DDL statement is rolled back. | Event notifications do not fire if the event in the corresponding DDL statement is rolled back. |

| SQL TRACE | EVENT NOTIFICATIONS |
|---|---|
| Managing the intermediate flow of trace event data involves populating and managing trace files or trace tables. | Intermediate management of event notification data is accomplished automatically through Service Broker queues. |
| Traces must be restarted every time the server restarts. | After being registered, event notifications persist across server cycles and are transacted. |
| After being initiated, the firing of traces cannot be controlled. Stop times and filter times can be used to specify when they initiate. Traces are accessed by polling the corresponding trace file. | Event notifications can be controlled by using the WAITFOR statement against the queue that receives the message generated by the event notification. They can be accessed by polling the queue. |
| ALTER TRACE is the least permission that is required to create a trace. Permission is also required to create a trace file on the corresponding computer. | Least permission depends on the type of event notification being created. RECEIVE permission is also needed on the corresponding queue. |
| Traces can be received remotely. | Event notifications can be received remotely. |
| Trace events are implemented by using system stored procedures. | Event notifications are implemented by using a combination of Database Engine and Service Broker Transact-SQL statements. |
| Trace event data can be accessed programmatically by querying the corresponding trace table, parsing the trace file, or using the SQL Server Management Objects (SMO) TraceReader Class. | Event data is accessed programmatically by issuing XQuery against the XML-formatted event data, or by using the SMO Event classes. |

## Event Notification Tasks

| TASK | TOPIC |
|---|---|
| Describes how to create and implement event notifications. | Implement Event Notifications |
| Describes how to configure Service Broker dialog security for event notifications that send messages to a service broker on a remote server. | Configure Dialog Security for Event Notifications |
| Describes how to return information about event notifications. | Get Information About Event Notifications |

## See Also

DDL Triggers
DML Triggers
SQL Trace

# Showplan Logical and Physical Operators Reference

5/3/2018 • 37 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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. You can view the query plan by using the SET SHOWPLAN statements, the graphical execution plan options in SQL Server Management Studio, or the SQL Server Profiler Showplan event classes.

Operators are classified as logical and 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. Each physical operator is an object or routine that performs an operation. For example, some physical operators access columns or rows from a table, index or view. Other physical operators perform other operations such as calculations, aggregations, data integrity checks or joins. Physical operators have costs associated with them.

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. For more information about these SET options, see SET STATISTICS PROFILE (Transact-SQL) and SET STATISTICS XML (Transact-SQL).

  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. If an operator is on the inner side of a loop join, the sum of the number of rebinds and rewinds should equal the number of rows processed on the outer side of the join. A rebind means that one or more of the correlated parameters of the join changed and the inner side must be reevaluated. A rewind means that none of the correlated parameters changed and the prior inner result set may be reused.

  **ActualRebinds** and **ActualRewinds** are present in XML Showplan output produced by using SET STATISTICS XML ON. They are only populated for the **Nonclustered Index Spool**, **Remote Query**, **Row Count Spool**, **Sort**, **Table Spool**, and **Table-valued Function** operators. **ActualRebinds** and **ActualRewinds** may also be populated for the **Assert** and **Filter** operators when the **StartupExpression**

attribute is set to TRUE.

When **ActualRebinds** and **ActualRewinds** are present in an XML Showplan, they are comparable to **EstimateRebinds** and **EstimateRewinds**. When they are absent, the estimated number of rows (**EstimateRows**) is comparable to the actual number of rows (**ActualRows**). Note that actual graphical Showplan output displays zeros for the actual rebinds and actual rewinds when they are absent.

A related counter, **ActualEndOfScans**, is available only when Showplan output is produced by using SET STATISTICS XML ON. Whenever a physical operator reaches the end of its data stream, this counter is incremented by one. A physical operator can reach the end of its data stream zero, one, or multiple times. As with rebinds and rewinds, the number of end of scans can be more than one only if the operator is on the inner side of a loop join. The number of end of scans should be less than or equal to the sum of the number of rebinds and rewinds.

## Mapping Physical and Logical Operators

The query optimizer creates a query plan as a tree consisting of logical operators. After the query optimizer creates the plan, the query optimizer chooses the most efficient physical operator for each logical operator. The query optimizer uses a cost-based approach to determine which physical operator will implement a logical operator.

Usually, a logical operation can be implemented by multiple physical operators. However, in rare cases, a physical operator can implement multiple logical operations as well.

## Operator Descriptions

This section contains descriptions of the logical and physical operators.

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
| --- | --- | --- |
| | **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. |
| A / (B+C) | **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. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| | **Assert** | The **Assert** operator verifies a condition. For example, it validates referential integrity or ensures that a scalar subquery returns one row. For each input row, the **Assert** operator evaluates the expression in the **Argument** column of the execution plan. If this expression evaluates to NULL, the row is passed through the **Assert** operator and the query execution continues. If this expression evaluates to a nonnull value, the appropriate error will be raised. The **Assert** operator is a physical operator. |
| | **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. **Async Concat** issues `open()` calls to all of the children simultaneously and then applies a bitmap to each child. For each bit that is a 1, **Async Concat** sends the output rows to the parent node on demand. |
| | **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. A bitmap filter uses a compact representation of a set of values from a table in one part of the operator tree to filter rows from a second table in another part of the tree. By removing unnecessary rows early in the query, subsequent operators have fewer rows to work with, and the overall performance of the query improves. The optimizer determines when a bitmap is selective enough to be useful and in which operators to apply the filter. **Bitmap** is a physical operator. |
| | **Bitmap Create** | The **Bitmap Create** operator appears in the Showplan output where bitmaps are built. **Bitmap Create** is a logical operator. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
| --- | --- | --- |
| | **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. The **Argument** column contains the bookmark label used to look up the row in the table or clustered index. The **Argument** column also contains the name of the table or clustered index in which the row is looked up. 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 table or clustered index.<br><br>**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** | 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. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
| --- | --- | --- |
| | **Clustered Index Delete** | 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** | The **Clustered Index Insert** Showplan operator inserts rows from its input into the clustered index specified in the Argument column. The Argument column also contains a SET:() predicate, which indicates the value to which each column is set. If **Clustered Index Insert** has no children for insert values, the row inserted is taken from the **Insert** operator itself.**Clustered Index Insert** is a physical 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** | 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. If the ORDERED clause is not present, the storage engine scans the index in the optimal way, without necessarily sorting the output. **Clustered Index Scan** is a logical and physical operator. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
| --- | --- | --- |
| | **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.<br><br>If the **Argument** column contains the ORDERED clause, the query processor has determined that the rows must be returned in the order in which the clustered index has sorted them. If the ORDERED clause is not present, the storage engine searches the index in the optimal way, without necessarily sorting the output. Allowing the output to retain its ordering can be less efficient than producing nonsorted output. When the keyword LOOKUP appears, then a bookmark lookup is being performed. In SQL Server 2008 and later versions, the **Key Lookup** operator provides bookmark lookup functionality. **Clustered Index Seek** is a logical and physical operator. |
| | **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. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| | **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. The **Argument** column contains a GROUP BY:() clause that specifies a list of key columns. If the query processor encounters adjacent rows that delete and insert the same key values, it replaces these separate operations with a single more efficient update operation. **Collapse** is a logical and physical operator. |
| | **Columnstore Index Scan** | The **Columnstore Index Scan** operator scans the columnstore index specified in the **Argument** column of the query execution plan. |
| | **Compute Scalar** | The **Compute Scalar** operator evaluates an expression to produce a computed scalar value. This may then be returned to the user, referenced elsewhere in the query, or both. An example of both is in a filter predicate or join predicate. **Compute Scalar** is a logical and physical operator.<br><br>**Compute Scalar** operators that appear in Showplans generated by SET STATISTICS XML might not contain the **RunTimeInformation** element. In graphical Showplans, **Actual Rows**, **Actual Rebinds**, and **Actual Rewinds** might be absent from the **Properties** window when the **Include Actual Execution Plan** option is selected in SQL Server Management Studio. When this occurs, it means that although these operators were used in the compiled query plan, their work was performed by other operators in the run-time query plan. Also note that the number of executes in Showplan output generated by SET STATISTICS PROFILE is equivalent to the sum of rebinds and rewinds in Showplans generated by SET STATISTICS XML. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| | **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** | 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. |
| | **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. |
| | **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). |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| 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.<br><br>Logical operators include Asynchronous, Optimistic, Primary, Read Only, Scroll Locks, and Secondary and Synchronous.<br><br>Physical operators include Dynamic, Fetch Query, Keyset, Population Query, Refresh Query and Snapshot. |
|  | **Declare** | The **Declare** operator allocates a local variable in the query plan. **Declare** is a language element. |
|  | **Delete** | The **Delete** operator deletes from an object rows that satisfy the optional predicate in the **Argument** column. |
|  | **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. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| | **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. Each record from the input stream appears in one of the output streams. 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.<br><br>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** | The **Dynamic** operator uses a cursor that can see all changes made by others. |
| | **Eager Spool** | 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** | The **Fetch Query** operator retrieves rows when a fetch is issued against a cursor. |
| | **Filter** | The **Filter** operator scans the input, returning only those rows that satisfy the filter expression (predicate) that appears in the **Argument** column. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| None | **Flow Distinct** | 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 | **Full Outer Join** | 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:<br><br>-The first input that had no matches in the second input.<br><br>-The second input that had no matches in the first input.<br><br>The input that does not contain the matching values is returned as a null value. **Full Outer Join** is a logical operator. |
|  | **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. If the output is ordered, the **Argument** column contains an ORDER BY:() predicate and the names of columns being ordered. **Gather Streams** is a logical operator. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| | **Hash Match** | The **Hash Match** operator builds a hash table by computing a hash value for each row from its build input. A HASH:() predicate with a list of columns used to create a hash value appears in the **Argument** column. Then, for each probe row (as applicable), it computes a hash value (using the same hash function) and looks in the hash table for matches. If a residual predicate is present (identified by RESIDUAL:() in the **Argument** column), that predicate must also be satisfied for rows to be considered a match. Behavior depends on the logical operation being performed:<br><br>-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.<br><br>-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.<br><br>-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.<br><br>**Hash Match** is a physical operator. |
| | **If** | 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** | 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. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| | **Inserted Scan** | The **Inserted Scan** operator scans the **inserted** table. **Inserted Scan** is a logical and physical operator. |
| | **Intrinsic** | The **Intrinsic** operator invokes an internal Transact-SQL function. **Intrinsic** is a language element. |
| | **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). |
| | **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.<br><br>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** | The **Keyset** operator uses a cursor that can see updates, but not inserts made by others. |
| | **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). |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
| --- | --- | --- |
| | **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 first (top) input when there is no matching row in the second (bottom) 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** | The **Log Row Scan** operator scans the transaction log. **Log Row Scan** is a logical and physical operator. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| | **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** | 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.<br><br>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** | 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. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| | **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. |
| | **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. |
| | **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. |
| | **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. It optionally may include a WHERE:() predicate, which the storage engine will evaluate against all rows that satisfy the SEEK:() predicate (it does not use the indexes to do this). If the **Argument** column contains the ORDERED clause, the query processor has determined that the rows must be returned in the order in which the nonclustered index has sorted them. If the ORDERED clause is not present, the storage engine searches the index in the optimal way (which does not guarantee that the output will be sorted). Allowing the output to retain its ordering may be less efficient than producing nonsorted output. **Index Seek** is a logical and physical operator. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| | **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** | 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** | 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. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| None | **Parallelism** | The **Parallelism** operator (or Exchange Iterator) performs the distribute streams, gather streams, and repartition streams logical operations. The **Argument** columns can contain a PARTITION COLUMNS:() predicate with a comma-separated list of the columns being partitioned. The **Argument** columns can also contain an ORDER BY:() predicate, listing the columns to preserve the sort order for during partitioning. **Parallelism** is a physical operator. For more information about the Parallelism operator, see Craig Freedman's blog series.<br><br>**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** | 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** | The **Population Query** operator populates the work table of a cursor when the cursor is opened. |
|  | **Refresh Query** | The **Refresh Query** operator fetches current data for rows in the fetch buffer. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| | **Remote Delete** | The **Remote Delete** operator deletes the input rows from a remote object. **Remote Delete** is a logical and physical 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** | 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** | The **Remote Insert** operator inserts the input rows into a remote object. **Remote Insert** is a logical and physical operator. |
| | **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** | 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** | The **Remote Update** operator updates the input rows in a remote object. **Remote Update** is a logical and physical operator. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| | **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** | 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** | **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" 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. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| 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** | 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** | **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. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
| --- | --- | --- |
| 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** | 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** | 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** | The **Snapshot** operator creates a cursor that does not see changes made by others. |
| | **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. The columns are prefixed with the value ASC if the columns are sorted in ascending order, or the value DESC if the columns are sorted in descending order. **Sort** is a logical and physical operator. |
| | **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. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| | **Spool** | The **Spool** operator saves an intermediate query result to the **tempdb** database. |
| | **Stream Aggregate** | The **Stream Aggregate** operator groups rows by one or more columns and then calculates one or more aggregate expressions returned by the query. The output of this operator can be referenced by later operators in the query, returned to the client, or both. The **Stream Aggregate** operator requires input ordered by the columns within its groups. The optimizer will use a **Sort** operator prior to this operator if the data is not already sorted due to a prior **Sort** operator or due to an ordered index seek or scan. In the SHOWPLAN_ALL statement or the graphical execution plan in SQL Server Management Studio, the columns in the GROUP BY predicate are listed in the **Argument** column, and the aggregate expressions are listed in the **Defined Values** column. **Stream Aggregate** is a physical operator. |
| | **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** | The **Table Delete** physical operator deletes rows from the table specified in the **Argument** column of the query execution plan. |
| | **Table Insert** | The **Table Insert** operator inserts rows from its input into the table specified in the **Argument** column of the query execution plan. The **Argument** column also contains a SET:() predicate, which indicates the value to which each column is set. If **Table Insert** has no children for insert values, then the row inserted is taken from the Insert operator itself. **Table Insert** is a physical operator. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
| --- | --- | --- |
| | **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. The actual operation performed depends on the run-time value of the **ACTION** column specified in the **Argument** column of the operator. **Table Merge** is a physical operator. |
| | **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** | 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** | 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. These values may be referenced in the SET clause or elsewhere within this operator as well as elsewhere within this query. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
|---|---|---|
| | **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**.<br><br>Queries with calls to table-valued functions generate query plans with the **Table-valued Function** iterator. **Table-valued Function** can be evaluated with different parameter values:<br><br>-<br>**Table-valued Function XML Reader** inputs an XML BLOB as a parameter and produces a rowset representing XML nodes in XML document order. Other input parameters may restrict XML nodes returned to a subset of XML document.<br><br>-**Table Valued Function XML Reader with XPath filter** is a special type of **XML Reader Table-valued Function** that restricts output to XML nodes satisfying an XPath expression.<br><br>**Table-valued Function** is a logical and physical operator. |
| | **Top** | The **Top** operator scans the input, returning only the first specified number or percent of rows, possibly based on a sort order. The **Argument** column can contain a list of the columns that are being checked for ties. In update plans, the **Top** operator is used to enforce row count limits. **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. |
| | **UDX** | Extended Operators (UDX) implement one of many XQuery and XPath operations in SQL Server. All UDX operators are both logical and physical operators.<br><br>Extended operator (UDX) **FOR XML** is used to serialize the relational row set it |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
| --- | --- | --- |
| | | inputs into XML representation in a single BLOB column in a single output row. It is an order sensitive XML aggregation operator.<br><br>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.<br><br>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.<br><br>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.<br><br>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.<br><br>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()**.<br><br>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()**.<br><br>Extended operator **UPDATE XML NODE** updates XML node in the XQuery replace data modification extension in the **modify()** method on XML type. |

| GRAPHICAL EXECUTION PLAN ICON | SHOWPLAN OPERATOR | DESCRIPTION |
| --- | --- | --- |
| None | **Union** | The **Union** operator scans multiple inputs, outputting each row scanned and removing duplicates. **Union** is a logical operator. |
| | **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** | The **While** operator implements the Transact-SQL while loop. **While** is a language element |
| | **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. |

# Spatial Data (SQL Server)

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Spatial data represents information about the physical location and shape of geometric objects. These objects can be point locations or more complex objects such as countries, roads, or lakes.

SQL Server supports two spatial data types: the **geometry** data type and the **geography** data type.

- The **geometry** type represents data in a Euclidean (flat) coordinate system.
- The **geography** type represents data in a round-earth coordinate system.

  Both data types are implemented as .NET common language runtime (CLR) data types in SQL Server.

> **IMPORTANT**
> For a detailed description and examples of spatial features introduced in SQL Server 2012 (11.x), download the white paper, New Spatial Features in SQL Server 2012.

## Related Tasks

Create, Construct, and Query geometry Instances
Describes the methods that you can use with instances of the geometry data type.

Create, Construct, and Query geography Instances
Describes the methods that you can use with instances of the geography data type.

Query Spatial Data for Nearest Neighbor
Describes the common query pattern that is used to find the closest spatial objects to a specific spatial object.

Create, Modify, and Drop Spatial Indexes
Provides information about creating, altering, and dropping a spatial index.

## Related Content

Spatial Data Types Overview
Introduces the spatial data types.

- Point
- LineString
- CircularString
- CompoundCurve
- Polygon
- CurvePolygon
- MultiPoint
- MultiLineString

- MultiPolygon
- GeometryCollection

Spatial Indexes Overview

Introduces spatial indexes and describes tessellation and tessellation schemes.

# SQL Trace

5/3/2018 • 10 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

In SQL Trace, events are gathered if they are instances of event classes listed in the trace definition. These events can be filtered out of the trace or queued for their destination. The destination can be a file or SQL Server Management Objects (SMO), which can use the trace information in applications that manage SQL Server.

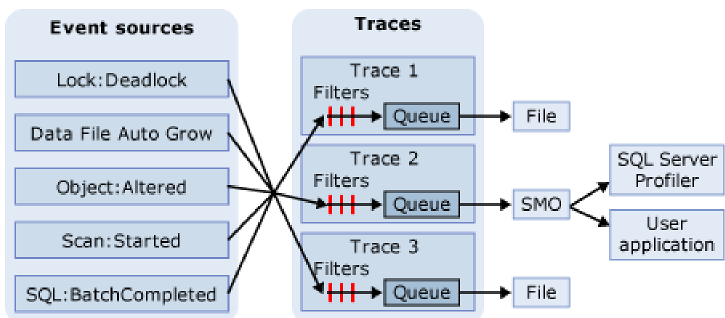
> **IMPORTANT**
>
> This feature will be removed in a future version of Microsoft SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use this feature. Use Extended Events instead.

## Benefits of SQL Trace

Microsoft SQL Server provides Transact-SQL system stored procedures to create traces on an instance of the SQL Server Database Engine. These system stored procedures can be used from within your own applications to create traces manually, instead of using SQL Server Profiler. This allows you to write custom applications specific to the needs of your enterprise.

## SQL Trace Architecture

Event Sources can be any source that produces the trace event, such as Transact-SQL batches or SQL Server events, such as deadlocks. For more information about events, see SQL Server Event Class Reference. After an event occurs, if the event class has been included in a trace definition, the event information is gathered by the trace. If filters have been defined for the event class in the trace definition, the filters are applied and the trace event information is passed to a queue. From the queue, the trace information is either written to a file or can be used by SMO in applications, such as SQL Server Profiler. The following diagram shows how SQL Trace gathers events during a tracing.




## SQL Trace Terminology

The following terms describe the key concepts of SQL Trace.

**Event**
The occurrence of an action within an instance of the Microsoft SQL Server Database Engine.

**Data column**
An attribute of an event.

**Event class**
A type of event that can be traced. The event class contains all of the data columns that can be reported by an event.

**Event category**
A group of related event classes.

**Trace** (noun)
A collection of events and data returned by the Database Engine.

**Trace** (verb)
To collect and monitor events in an instance of SQL Server.

**Tracedefinition**
A collection of event classes, data columns and filters that identify the types of events to be collected during a trace.

**Filter**
Criteria that limit the events that are collected in a trace.

**Trace file**
A file created when a trace is saved.

**Template**
In SQL Server Profiler, a file that defines the event classes and data columns to be collected in a trace.

**Trace table**
In SQL Server Profiler, a table that is created when a trace is saved to a table.

# Use Data Columns to Describe Returned Events

SQL Trace uses data columns in the trace output to describe events that are returned when the trace runs. The following table describes the SQL Server Profiler data columns, which are the same data columns as those used by SQL Trace, and indicates the columns that are selected by default.

| DATA COLUMN | COLUMN NUMBER | DESCRIPTION |
| --- | --- | --- |
| **ApplicationName*** | 10 | The name of the client application that created the connection to an instance of SQL Server. This column is populated with the values passed by the application and not the name of the program. |
| **BigintData1** | 52 | Value (**bigint** data type), which depends on the event class specified in the trace. |
| **BigintData2** | 53 | Value (**bigint** data type), which depends on the event class specified in the trace. |
| **Binary Data** | 2 | The binary value dependent on the event class that is captured in the trace. |
| **ClientProcessID*** | 9 | The ID assigned by the host computer to the process where the client application is running. This data column is populated if the client process ID is provided by the client. |

| DATA COLUMN | COLUMN NUMBER | DESCRIPTION |
|---|---|---|
| **ColumnPermissions** | 44 | Indicates whether a column permission was set. You can parse the statement text to determine which permissions were applied to which columns. |
| **CPU** | 18 | The amount of CPU time (in milliseconds) that is used by the event. |
| **Database ID*** | 3 | The ID of the database specified by the USE *database_name* statement, or the ID of the default database if no USE *database_name*statement has been issued for a given instance. SQL Server Profiler displays the name of the database if the **ServerName** data column is captured in the trace and the server is available. Determine the value for a database by using the DB_ID function. |
| **DatabaseName** | 35 | The name of the database in which the user statement is running. |
| **DBUserName*** | 40 | The SQL Server user name of the client. |
| **Duration** | 13 | The duration (in microseconds) of the event.<br><br>The server reports the duration of an event in microseconds (one millionth, or $10^{-6}$, of a second) and the amount of CPU time used by the event in milliseconds (one thousandth, or $10^{-3}$, of a second). The SQL Server Profiler graphical user interface displays the **Duration** column in milliseconds by default, but when a trace is saved to either a file or a database table, the **Duration** column value is written in microseconds. |
| **EndTime** | 15 | The time at which the event ended. This column is not populated for event classes that refer to an event that is starting, such as **SQL:BatchStarting** or **SP:Starting**. |
| **Error** | 31 | The error number of a given event. Often this is the error number stored in **sysmessages**. |
| **EventClass*** | 27 | The type of event class that is captured. |
| **EventSequence** | 51 | Sequence number for this event. |

| DATA COLUMN | COLUMN NUMBER | DESCRIPTION |
| --- | --- | --- |
| **EventSubClass*** | 21 | The type of event subclass, which provides further information about each event class. For example, event subclass values for the **Execution Warning** event class represent the type of execution warning:<br><br>**1** = Query wait. The query must wait for resources before it can execute; for example, memory.<br><br>**2** = Query time-out. The query timed out while waiting for required resources to execute. This data column is not populated for all event classes. |
| **GUID** | 54 | GUID value which depends on the event class specified in the trace. |
| **FileName** | 36 | The logical name of the file that is modified. |
| **Handle** | 33 | The integer used by ODBC, OLE DB, or DB-Library to coordinate server execution. |
| **HostName*** | 8 | The name of the computer on which the client is running. This data column is populated if the host name is provided by the client. To determine the host name, use the HOST_NAME function. |
| **IndexID** | 24 | The ID for the index on the object affected by the event. To determine the index ID for an object, use the **indid** column of the **sysindexes** system table. |
| **IntegerData** | 25 | The integer value dependent on the event class captured in the trace. |
| **IntegerData2** | 55 | The integer value dependent on the event class captured in the trace. |
| **IsSystem** | 60 | Indicates whether the event occurred on a system process or a user process:<br><br>**1** = system<br><br>**0** = user |
| **LineNumber** | 5 | Contains the number of the line that contains the error. For events that involve Transact-SQL statements, like **SP:StmtStarting**, the **LineNumber** contains the line number of the statement in the stored procedure or batch. |

| DATA COLUMN | COLUMN NUMBER | DESCRIPTION |
| --- | --- | --- |
| **LinkedServerName** | 45 | Name of the linked server. |
| **LoginName** | 11 | The name of the login of the user (either SQL Server security login or the Windows login credentials in the form of DOMAIN\Username). |
| **LoginSid*** | 41 | The security identifier (SID) of the logged-in user. You can find this information in the **sys.server_principals** view of the **master** database. Each login to the server has a unique ID. |
| **MethodName** | 47 | Name of the OLEDB method. |
| **Mode** | 32 | The integer used by various events to describe a state the event is requesting or has received. |
| **NestLevel** | 29 | The integer that represents the data returned by @@NESTLEVEL. |
| **NTDomainName*** | 7 | The Microsoft Windows domain to which the user belongs. |
| **NTUserName*** | 6 | The Windows user name. |
| **ObjectID** | 22 | The system-assigned ID of the object. |
| **ObjectID2** | 56 | The ID of the related object or entity, if available. |
| **ObjectName** | 34 | The name of the object that is referenced. |
| **ObjectType**** | 28 | The value representing the type of the object involved in the event. This value corresponds to the **type** column in **sysobjects**. |
| **Offset** | 61 | The starting offset of the statement within the stored procedure or batch. |
| **OwnerID** | 58 | For lock events only. The type of the object that owns a lock. |
| **OwnerName** | 37 | The database user name of the object owner. |
| **ParentName** | 59 | The name of the schema in which the object resides. |

| DATA COLUMN | COLUMN NUMBER | DESCRIPTION |
| --- | --- | --- |
| **Permissions** | 19 | The integer value that represents the type of permissions checked. Values are:<br><br>**1** = SELECT ALL<br><br>**2** = UPDATE ALL<br><br>**4** = REFERENCES ALL<br><br>**8** = INSERT<br><br>**16** = DELETE<br><br>**32** = EXECUTE (procedures only)<br><br>**4096** = SELECT ANY (at least one column)<br><br>**8192** = UPDATE ANY<br><br>**16384** = REFERENCES ANY |
| **ProviderName** | 46 | Name of the OLEDB provider. |
| **Reads** | 16 | The number of read operations on the logical disk that are performed by the server on behalf of the event. These read operations include all reads from tables and buffers during the statement's execution. |
| **RequestID** | 49 | ID of the request that contains the statement. |
| **RoleName** | 38 | The name of the application role that is being enabled. |
| **RowCounts** | 48 | The number of rows in the batch. |
| **ServerName***| 26 | The name of the instance of SQL Server that is being traced. |
| **SessionLoginName** | 64 | The login name of the user who originated the session. For example, if you connect to SQL Server using **Login1** and execute a statement as **Login2**, **SessionLoginName** displays **Login1**, while **LoginName** displays **Login2**. This data column displays both SQL Server and Windows logins. |
| **Severity** | 20 | The severity level of the exception event. |
| **SourceDatabaseID** | 62 | The ID of the database in which the source of the object exists. |

| DATA COLUMN | COLUMN NUMBER | DESCRIPTION |
|---|---|---|
| **SPID** | 12 | The server process ID (SPID) that is assigned by SQL Server to the process that is associated with the client. |
| **SqlHandle** | 63 | 64-bit hash based on the text of an ad hoc query or the database and object ID of an SQL object. This value can be passed to **sys.dm_exec_sql_text()** to retrieve the associated SQL text. |
| **StartTime*** | 14 | The time at which the event started, when available. |
| **State** | 30 | Error state code. |
| **Success** | 23 | Represents whether the event was successful. Values include:<br><br>**1** = Success.<br><br>**0** = Failure<br><br>For example, a **1** means a successful permissions check, and a **0** means a failed check. |
| **TargetLoginName** | 42 | For actions that target a login, the name of the targeted login; for example, to add a new login. |
| **TargetLoginSid** | 43 | For actions that target a login, the SID of the targeted login; for example, to add a new login. |
| **TargetUserName** | 39 | For actions that target a database user, the name of that user; for example, to grant permission to a user. |
| **TextData** | 1 | The text value dependent on the event class that is captured in the trace. However, if you trace a parameterized query, the variables are not displayed with data values in the **TextData** column. |
| **Transaction ID** | 4 | The system-assigned ID of the transaction. |
| **Type** | 57 | The integer value dependent on the event class captured in the trace. |
| **Writes** | 17 | The number of physical disk write operations that are performed by the server on behalf of the event. |

| DATA COLUMN | COLUMN NUMBER | DESCRIPTION |
|---|---|---|
| **XactSequence** | 50 | A token to describe the current transaction. |

*These data columns are populated by default for all events.

**For more information about the **ObjectType** data column, see ObjectType Trace Event Column.

## SQL Trace Tasks

| TASK DESCRIPTION | TOPIC |
|---|---|
| Describes how to create and run traces using Transact-SQL stored procedures. | Create and Run Traces Using Transact-SQL Stored Procedures |
| Describes how to create manual traces using stored procedures on an instance of the SQL Server Database Engine. | Create Manual Traces using Stored Procedures |
| Describes how to save trace results to the file where the trace results are written. | Save Trace Results to a File |
| Describes how to improve access to trace data by using space in the **temp** directory. | Improve Access to Trace Data |
| Describes how to use stored procedures to create a trace. | Create a Trace (Transact-SQL) |
| Describes how to use stored procedures to create a filter that retrieves only the information you need on an event being traced. | Set a Trace Filter (Transact-SQL) |
| Describes how to use stored procedures to modify an existing trace. | Modify an Existing Trace (Transact-SQL) |
| Describes how to use built-in functions to view a saved trace. | View a Saved Trace (Transact-SQL) |
| Describes how to use built-in functions to view trace filter information. | View Filter Information (Transact-SQL) |
| Describes how to use stored procedures to delete a trace. | Delete a Trace (Transact-SQL) |
| Describes how to minimize the performance cost incurred by a trace. | Optimize SQL Trace |
| Describes how to filter a trace to minimize the overhead that is incurred during a trace. | Filter a Trace |
| Describes how to minimize the amount of data that the trace collects. | Limit Trace File and Table Sizes |
| Describes the two ways to schedule tracing in Microsoft SQL Server. | Schedule Traces |

## See Also

SQL Server Profiler Templates and Permissions

SQL Server Management Objects (SMO) Programming Guide

# Statistics

5/3/2018 • 26 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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. This topic discusses statistics concepts and provides guidelines for using query optimization statistics effectively.

## Components and Concepts

### Statistics

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.

To create the histogram, the query optimizer sorts the column values, computes the number of values that match each distinct column value and then aggregates the column values into a maximum of 200 contiguous histogram steps. Each histogram step includes a range of column values followed by an upper bound column value. The range includes all possible column values between boundary values, excluding the boundary values themselves. The lowest of the sorted column values is the upper boundary value for the first histogram step.

In more detail, SQL Server creates the **histogram** from the sorted set of column values in three steps:

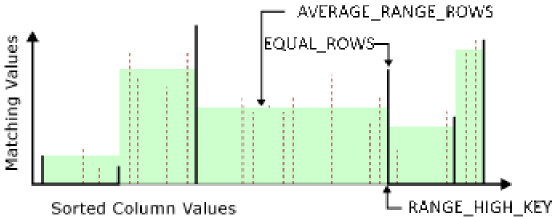
- **Histogram initialization**: In the first step, a sequence of values starting at the beginning of the sorted set is processed, and up to 200 values of *range_high_key*, *equal_rows*, *range_rows*, and *distinct_range_rows* are collected (*range_rows* and *distinct_range_rows* are always zero during this step). The first step ends either when all input has been exhausted, or when 200 values have been found.
- **Scan with bucket merge**: Each additional value from the leading column of the statistics key is processed in

the second step, in sorted order; each successive value is either added to the last range or a new range at the end is created (this is possible because the input values are sorted). If a new range is created, then one pair of existing, neighboring ranges is collapsed into a single range. This pair of ranges is selected in order to minimize information loss. This method uses a *maximum difference* algorithm to minimize the number of steps in the histogram while maximizing the difference between the boundary values. The number of steps after collapsing ranges stays at 200 throughout this step.

- **Histogram consolidation**: In the third step, more ranges may be collapsed if a significant amount of information is not lost. The number of histogram steps can be fewer than the number of distinct values, even for columns with fewer than 200 boundary points. Therefore, even if the column has more than 200 unique values, the histogram may have less than 200 steps. For a column consisting of only unique values, then the consolidated histogram will have a minimum of three steps.

> **NOTE**
> If the histogram has been built using a sample rather than fullscan, then the values of *equal_rows*, *range_rows*, and *distinct_range_rows* and *average_range_rows* are estimated, and therefore they do not need to be whole integers.

The following diagram shows a histogram with six steps. The area to the left of the first upper boundary value is the first step.




For each histogram step above:

- Bold line represents the upper boundary value (*range_high_key*) and the number of times it occurs (*equal_rows*)

- Solid area left of *range_high_key* represents the range of column values and the average number of times each column value occurs (*average_range_rows*). The *average_range_rows* for the first histogram step is always 0.

- Dotted lines represent the sampled values used to estimate total number of distinct values in the range (*distinct_range_rows*) and total number of values in the range (*range_rows*). The query optimizer uses *range_rows* and *distinct_range_rows* to compute *average_range_rows* and does not store the sampled values.

#### Density Vector

**Density** is information about the number of duplicates in a given column or combination of columns and it is calculated as 1/(number of distinct values). The query optimizer uses densities to enhance cardinality estimates for queries that return multiple columns from the same table or indexed view. The density vector contains one density for each prefix of columns in the statistics object.

> **NOTE**
> Frequency is information about the occurrence of each distinct value in the first key column of the statistics object, and is calculated as row count * density. A maximum frequency of 1 can be found in columns with unique values.

For example, if a statistics object has the key columns `CustomerId`, `ItemId` and `Price`, density is calculated on

each of the following column prefixes.

| COLUMN PREFIX | DENSITY CALCULATED ON |
| --- | --- |
| (CustomerId) | Rows with matching values for CustomerId |
| (CustomerId, ItemId) | Rows with matching values for CustomerId and ItemId |
| (CustomerId, ItemId, Price) | Rows with matching values for CustomerId, ItemId, and Price |

### Filtered Statistics

Filtered statistics can improve query performance for queries that select from well-defined subsets of data. Filtered statistics use a filter predicate to select the subset of data that is included in the statistics. Well-designed filtered statistics can improve the query execution plan compared with full-table statistics. For more information about the filter predicate, see CREATE STATISTICS (Transact-SQL). For more information about when to create filtered statistics, see the When to Create Statistics section in this topic.

### Statistics Options

There are three options that you can set that affect when and how statistics are created and updated. These options are set at the database level only.

#### AUTO_CREATE_STATISTICS Option

When the automatic create statistics option, AUTO_CREATE_STATISTICS is ON, the Query Optimizer creates statistics on individual columns in the query predicate, as necessary, to improve cardinality estimates for the query plan. These single-column statistics are created on columns that do not already have a histogram in an existing statistics object. The AUTO_CREATE_STATISTICS option does not determine whether statistics get created for indexes. This option also does not generate filtered statistics. It applies strictly to single-column statistics for the full table.

When the Query Optimizer creates statistics as a result of using the AUTO_CREATE_STATISTICS option, the statistics name starts with `_WA`. You can use the following query to determine if the Query Optimizer has created statistics for a query predicate column.

```
SELECT OBJECT_NAME(s.object_id) AS object_name,
    COL_NAME(sc.object_id, sc.column_id) AS column_name,
    s.name AS statistics_name
FROM sys.stats AS s
INNER JOIN sys.stats_columns AS sc
    ON s.stats_id = sc.stats_id AND s.object_id = sc.object_id
WHERE s.name like '_WA%'
ORDER BY s.name;
```

#### AUTO_UPDATE_STATISTICS Option

When the automatic update statistics option, AUTO_UPDATE_STATISTICS is ON, the Query Optimizer determines when statistics might be out-of-date and then updates them when they are used by a query. Statistics become out-of-date after insert, update, delete, or merge operations change the data distribution in the table or indexed view. The Query Optimizer determines when statistics might be out-of-date by counting the number of data modifications since the last statistics update and comparing the number of modifications to a threshold. The threshold is based on the number of rows in the table or indexed view.

- Up to SQL Server 2014 (12.x), SQL Server uses a threshold based on the percent of rows changed. This is regardless of the number of rows in the table. The threshold is:
  - If the table cardinality was 500 or less at the time statistics were evaluated, update for every 500 modifications.

- If the table cardinality was above 500 at the time statistics were evaluated, update for every 500 + 20 percent of modifications.

- Starting with SQL Server 2016 (13.x) and under the database compatibility level 130, SQL Server uses a decreasing, dynamic statistics update threshold that adjusts according to the number of rows in the table. This is calculated as the square root of the product of 1000 and the current table cardinality. For example if your table contains 2 million rows, then the calculation is sqrt (1000 * 2000000) = 44721.359. With this change, statistics on large tables will be updated more often. However, if a database has a compatibility level below 130, then the SQL Server 2014 (12.x) threshold applies.

> **IMPORTANT**
>
> Starting with SQL Server 2008 R2 through SQL Server 2014 (12.x), or in SQL Server 2016 (13.x) through SQL Server 2017 under database compatibility level lower than 130, use trace flag 2371 and SQL Server will use a decreasing, dynamic statistics update threshold that adjusts according to the number of rows in the table.

The Query Optimizer checks for out-of-date statistics before compiling a query and before executing a cached query plan. Before compiling a query, the Query Optimizer uses the columns, tables, and indexed views in the query predicate to determine which statistics might be out-of-date. Before executing a cached query plan, the Database Engine verifies that the query plan references up-to-date statistics.

The AUTO_UPDATE_STATISTICS option applies to statistics objects created for indexes, single-columns in query predicates, and statistics created with the CREATE STATISTICS statement. This option also applies to filtered statistics.

For more information about controlling AUTO_UPDATE_STATISTICS, see Controlling Autostat (AUTO_UPDATE_STATISTICS) behavior in SQL Server.

**AUTO_UPDATE_STATISTICS_ASYNC**

The asynchronous statistics update option, AUTO_UPDATE_STATISTICS_ASYNC, determines whether the Query Optimizer uses synchronous or asynchronous statistics updates. By default, the asynchronous statistics update option is OFF, and the Query Optimizer updates statistics synchronously. The AUTO_UPDATE_STATISTICS_ASYNC option applies to statistics objects created for indexes, single columns in query predicates, and statistics created with the CREATE STATISTICS statement.

> **NOTE**
>
> To set the asynchronous statistics update option in SQL Server Management Studio, in the *Options* page of the *Database Properties* window, both *Auto Update Statistics* and *Auto Update Statistics Asynchronously* options need to be set to **True**.

Statistics updates can be either synchronous (the default) or asynchronous. With synchronous statistics updates, queries always compile and execute with up-to-date statistics; When statistics are out-of-date, the Query Optimizer waits for updated statistics before compiling and executing the query. With asynchronous statistics updates, queries compile with existing statistics even if the existing statistics are out-of-date; The Query Optimizer could choose a suboptimal query plan if statistics are out-of-date when the query compiles. Queries that compile after the asynchronous updates have completed will benefit from using the updated statistics.

Consider using synchronous statistics when you perform operations that change the distribution of data, such as truncating a table or performing a bulk update of a large percentage of the rows. If you do not update the statistics after completing the operation, using synchronous statistics will ensure statistics are up-to-date before executing queries on the changed data.

Consider using asynchronous statistics to achieve more predictable query response times for the following scenarios:

- Your application frequently executes the same query, similar queries, or similar cached query plans. Your query response times might be more predictable with asynchronous statistics updates than with synchronous statistics updates because the Query Optimizer can execute incoming queries without waiting for up-to-date statistics. This avoids delaying some queries and not others.
- Your application has experienced client request time outs caused by one or more queries waiting for updated statistics. In some cases, waiting for synchronous statistics could cause applications with aggressive time outs to fail.

**INCREMENTAL**

When INCREMENTAL option of CREATE STATISTICS is ON, the statistics created are per partition statistics. When OFF, the statistics tree is dropped and SQL Server re-computes the statistics. The default is OFF. This setting overrides the database level INCREMENTAL property. For more information about creating incremental statistics, see CREATE STATISTICS (Transact-SQL). For more information about creating per partition statistics automatically, see Database Properties (Options Page) and ALTER DATABASE SET Options (Transact-SQL).

When new partitions are added to a large table, statistics should be updated to include the new partitions. However the time required to scan the entire table (FULLSCAN or SAMPLE option) might be quite long. Also, scanning the entire table isn't necessary because only the statistics on the new partitions might be needed. The incremental option creates and stores statistics on a per partition basis, and when updated, only refreshes statistics on those partitions that need new statistics

If per partition statistics are not supported the option is ignored and a warning is generated. Incremental stats are not supported for following statistics types:

- Statistics created with indexes that are not partition-aligned with the base table.
- Statistics created on Always On readable secondary databases.
- Statistics created on read-only databases.
- Statistics created on filtered indexes.
- Statistics created on views.
- Statistics created on internal tables.
- Statistics created with spatial indexes or XML indexes.

**Applies to**: SQL Server 2014 (12.x) through SQL Server 2017.

## 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, see Create Filtered Indexes and CREATE INDEX (Transact-SQL).
2. The Query Optimizer creates statistics for single columns in query predicates when AUTO_CREATE_STATISTICS is on.

For most queries, these two methods for creating statistics ensure a high-quality query plan; in a few cases, you can improve query plans by creating additional statistics with the CREATE STATISTICS statement. These additional statistics can capture statistical correlations that the Query Optimizer does not account for when it creates statistics for indexes or single columns. Your application might have additional statistical correlations in the table data that, if calculated into a statistics object, could enable the Query Optimizer to improve query plans. For example, filtered statistics on a subset of data rows or multicolumn statistics on query predicate columns might improve the query plan.

When creating statistics with the CREATE STATISTICS statement, we recommend keeping the

AUTO_CREATE_STATISTICS option on so that the Query Optimizer continues to routinely create single-column statistics for query predicate columns. For more information about query predicates, see Search Condition (Transact-SQL).

Consider creating statistics with the CREATE STATISTICS statement when any of the following applies:

- The Database Engine Tuning Advisor suggests creating statistics.
- The query predicate contains multiple correlated columns that are not already in the same index.
- The query selects from a subset of data.
- The query has missing statistics.

**Query Predicate contains multiple correlated columns**

When a query predicate contains multiple columns that have cross-column relationships and dependencies, statistics on the multiple columns might improve the query plan. Statistics on multiple columns contain cross-column correlation statistics, called *densities*, that are not available in single-column statistics. Densities can improve cardinality estimates when query results depend on data relationships among multiple columns.

If the columns are already in the same index, the multicolumn statistics object already exists and it is not necessary to create it manually. If the columns are not already in the same index, you can create multicolumn statistics by creating an index on the columns or by using the CREATE STATISTICS statement. It requires more system resources to maintain an index than a statistics object. If the application does not require the multicolumn index, you can economize on system resources by creating the statistics object without creating the index.

When creating multicolumn statistics, the order of the columns in the statistics object definition affects the effectiveness of densities for making cardinality estimates. The statistics object stores densities for each prefix of key columns in the statistics object definition. For more information about densities, see Density section in this page.

To create densities that are useful for cardinality estimates, the columns in the query predicate must match one of the prefixes of columns in the statistics object definition. For example, the following creates a multicolumn statistics object on the columns `LastName`, `MiddleName`, and `FirstName`.

```
USE AdventureWorks2012;
GO
IF EXISTS (SELECT name FROM sys.stats
    WHERE name = 'LastFirst'
    AND object_ID = OBJECT_ID ('Person.Person'))
DROP STATISTICS Person.Person.LastFirst;
GO
CREATE STATISTICS LastFirst ON Person.Person (LastName, MiddleName, FirstName);
GO
```

In this example, the statistics object `LastFirst` has densities for the following column prefixes: `(LastName)`, `(LastName, MiddleName)`, and `(LastName, MiddleName, FirstName)`. The density is not available for `(LastName, FirstName)`. If the query uses `LastName` and `FirstName` without using `MiddleName`, the density is not available for cardinality estimates.

**Query Selects from a subset of data**

When the Query Optimizer creates statistics for single columns and indexes, it creates the statistics for the values in all rows. When queries select from a subset of rows, and that subset of rows has a unique data distribution, filtered statistics can improve query plans. You can create filtered statistics by using the CREATE STATISTICS statement with the WHERE clause to define the filter predicate expression.

For example, using AdventureWorks2012, each product in the `Production.Product` table belongs to one of four categories in the `Production.ProductCategory` table: Bikes, Components, Clothing, and Accessories. Each of the categories has a different data distribution for weight: bike weights range from 13.77 to 30.0, component weights

range from 2.12 to 1050.00 with some NULL values, clothing weights are all NULL, and accessory weights are also NULL.

Using Bikes as an example, filtered statistics on all bike weights will provide more accurate statistics to the Query Optimizer and can improve the query plan quality compared with full-table statistics or nonexistent statistics on the Weight column. The bike weight column is a good candidate for filtered statistics but not necessarily a good candidate for a filtered index if the number of weight lookups is relatively small. The performance gain for lookups that a filtered index provides might not outweigh the additional maintenance and storage cost for adding a filtered index to the database.

The following statement creates the `BikeWeights` filtered statistics on all of the subcategories for Bikes. The filtered predicate expression defines bikes by enumerating all of the bike subcategories with the comparison `Production.ProductSubcategoryID IN (1,2,3)`. The predicate cannot use the Bikes category name because it is stored in the Production.ProductCategory table, and all columns in the filter expression must be in the same table.

```
USE AdventureWorks2012;
GO
IF EXISTS ( SELECT name FROM sys.stats
    WHERE name = 'BikeWeights'
    AND object_ID = OBJECT_ID ('Production.Product'))
DROP STATISTICS Production.Product.BikeWeights;
GO
CREATE STATISTICS BikeWeights
    ON Production.Product (Weight)
WHERE ProductSubcategoryID IN (1,2,3);
GO
```

The Query Optimizer can use the `BikeWeights` filtered statistics to improve the query plan for the following query that selects all of the bikes that weigh more than `25`.

```
SELECT P.Weight AS Weight, S.Name AS BikeName
FROM Production.Product AS P
    JOIN Production.ProductSubcategory AS S
    ON P.ProductSubcategoryID = S.ProductSubcategoryID
WHERE P.ProductSubcategoryID IN (1,2,3) AND P.Weight > 25
ORDER BY P.Weight;
GO
```

#### Query identifies missing statistics

If an error or other event prevents the Query Optimizer from creating statistics, the Query Optimizer creates the query plan without using statistics. The Query Optimizer marks the statistics as missing and attempts to regenerate the statistics the next time the query is executed.

Missing statistics are indicated as warnings (table name in red text) when the execution plan of a query is graphically displayed using SQL Server Management Studio. Additionally, monitoring the **Missing Column Statistics** event class by using SQL Server Profiler indicates when statistics are missing. For more information, see Errors and Warnings Event Category (Database Engine).

If statistics are missing, perform the following steps:

- Verify that AUTO_CREATE_STATISTICS and AUTO_UPDATE_STATISTICS are on.
- Verify that the database is not read-only. If the database is read-only, a new statistics object cannot be saved.
- Create the missing statistics by using the CREATE STATISTICS statement.

When statistics on a read-only database or read-only snapshot are missing or stale, the Database Engine creates and maintains temporary statistics in **tempdb**. When the Database Engine creates temporary statistics, the statistics name is appended with the suffix *_readonly_database_statistic* to differentiate the temporary statistics

from the permanent statistics. The suffix *_readonly_database_statistic* is reserved for statistics generated by SQL Server. Scripts for the temporary statistics can be created and reproduced on a read-write database. When scripted, Management Studio changes the suffix of the statistics name from *_readonly_database_statistic* to *_readonly_database_statistic_scripted*.

Only SQL Server can create and update temporary statistics. However, you can delete temporary statistics and monitor statistics properties using the same tools that you use for permanent statistics:

- Delete temporary statistics using the DROP STATISTICS statement.
- Monitor statistics using the **sys.stats** and **sys.stats_columns** catalog views. **sys_stats** includes the **is_temporary** column, to indicate which statistics are permanent and which are temporary.

  Because temporary statistics are stored in **tempdb**, a restart of the SQL Server service causes all temporary statistics to disappear.

## When to update statistics

The Query Optimizer determines when statistics might be out-of-date and then updates them when they are needed for a query plan. In some cases you can improve the query plan and therefore improve query performance by updating statistics more frequently than occur when AUTO_UPDATE_STATISTICS is on. You can update statistics with the UPDATE STATISTICS statement or the stored procedure sp_updatestats.

Updating statistics ensures that queries compile with up-to-date statistics. However, updating statistics causes queries to recompile. We recommend not updating statistics too frequently because there is a performance tradeoff between improving query plans and the time it takes to recompile queries. The specific tradeoffs depend on your application.

When updating statistics with UPDATE STATISTICS or sp_updatestats, we recommend keeping AUTO_UPDATE_STATISTICS set to ON so that the Query Optimizer continues to routinely update statistics. For more information about how to update statistics on a column, an index, a table, or an indexed view, see UPDATE STATISTICS (Transact-SQL). For information about how to update statistics for all user-defined and internal tables in the database, see the stored procedure sp_updatestats (Transact-SQL).

To determine when statistics were last updated, use the sys.dm_db_stats_properties or STATS_DATE functions.

Consider updating statistics for the following conditions:

- Query execution times are slow.
- Insert operations occur on ascending or descending key columns.
- After maintenance operations.

**Query execution times are slow**

If query response times are slow or unpredictable, ensure that queries have up-to-date statistics before performing additional troubleshooting steps.

**Insert operations occur on ascending or descending key columns**

Statistics on ascending or descending key columns, such as IDENTITY or real-time timestamp columns, might require more frequent statistics updates than the Query Optimizer performs. Insert operations append new values to ascending or descending columns. The number of rows added might be too small to trigger a statistics update. If statistics are not up-to-date and queries select from the most recently added rows, the current statistics will not have cardinality estimates for these new values. This can result in inaccurate cardinality estimates and slow query performance.

For example, a query that selects from the most recent sales order dates will have inaccurate cardinality estimates if the statistics are not updated to include cardinality estimates for the most recent sales order dates.

**After maintenance operations**

Consider updating statistics after performing maintenance procedures that change the distribution of data, such as truncating a table or performing a bulk insert of a large percentage of the rows. This can avoid future delays in query processing while queries wait for automatic statistics updates.

Operations such as rebuilding, defragmenting, or reorganizing an index do not change the distribution of data. Therefore, you do not need to update statistics after performing ALTER INDEX REBUILD, DBCC DBREINDEX, DBCC INDEXDEFRAG, or ALTER INDEX REORGANIZE operations. The Query Optimizer updates statistics when you rebuild an index on a table or view with ALTER INDEX REBUILD or DBCC DBREINDEX, however this statistics update is a byproduct of re-creating the index. The Query Optimizer does not update statistics after DBCC INDEXDEFRAG or ALTER INDEX REORGANIZE operations.

> **TIP**
>
> Starting with SQL Server 2016 (13.x) SP1 CU4, use the PERSIST_SAMPLE_PERCENT option of CREATE STATISTICS (Transact-SQL) or UPDATE STATISTICS (Transact-SQL), to set and retain a specific sampling percentage for subsequent statistic updates that do not explicitly specify a sampling percentage.

## Queries that use statistics effectively

Certain query implementations, such as local variables and complex expressions in the query predicate, can lead to suboptimal query plans. Following query design guidelines for using statistics effectively can help to avoid this. For more information about query predicates, see Search Condition (Transact-SQL).

You can improve query plans by applying query design guidelines that use statistics effectively to improve *cardinality estimates* for expressions, variables, and functions used in query predicates. When the Query Optimizer does not know the value of an expression, variable, or function, it does not know which value to lookup in the histogram and therefore cannot retrieve the best cardinality estimate from the histogram. Instead, the Query Optimizer bases the cardinality estimate on the average number of rows per distinct value for all of the sampled rows in the histogram. This leads to suboptimal cardinality estimates and can hurt query performance. For more information about histograms, see histogram section in this page or sys.dm_db_stats_histogram.

The following guidelines describe how to write queries to improve query plans by improving cardinality estimates.

**Improving cardinality estimates for expressions**

To improve cardinality estimates for expressions, follow these guidelines:

- Whenever possible, simplify expressions with constants in them. The Query Optimizer does not evaluate all functions and expressions containing constants prior to determining cardinality estimates. For example, simplify the expression `ABS(-100)` to `100`.
- If the expression uses multiple variables, consider creating a computed column for the expression and then create statistics or an index on the computed column. For example, the query predicate `WHERE PRICE + Tax > 100` might have a better cardinality estimate if you create a computed column for the expression `Price + Tax`.

**Improving cardinality estimates for variables and functions**

To improve the cardinality estimates for variables and functions, follow these guidelines:

- 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.

  For example, the following stored procedure `Sales.GetRecentSales` changes the value of the parameter `@date` when `@date` is NULL.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ( 'Sales.GetRecentSales', 'P') IS NOT NULL
    DROP PROCEDURE Sales.GetRecentSales;
GO
CREATE PROCEDURE Sales.GetRecentSales (@date datetime)
AS BEGIN
    IF @date IS NULL
        SET @date = DATEADD(MONTH, -3, (SELECT MAX(ORDERDATE) FROM Sales.SalesOrderHeader))
    SELECT * FROM Sales.SalesOrderHeader h, Sales.SalesOrderDetail d
    WHERE h.SalesOrderID = d.SalesOrderID
    AND h.OrderDate > @date
END
GO
```

  If the first call to the stored procedure `Sales.GetRecentSales` passes a NULL for the `@date` parameter, the Query Optimizer will compile the stored procedure with the cardinality estimate for `@date = NULL` even though the query predicate is not called with `@date = NULL`. This cardinality estimate might be significantly different than the number of rows in the actual query result. As a result, the Query Optimizer might choose a suboptimal query plan. To help avoid this, you can rewrite the stored procedure into two procedures as follows:

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ( 'Sales.GetNullRecentSales', 'P') IS NOT NULL
    DROP PROCEDURE Sales.GetNullRecentSales;
GO
CREATE PROCEDURE Sales.GetNullRecentSales (@date datetime)
AS BEGIN
    IF @date is NULL
        SET @date = DATEADD(MONTH, -3, (SELECT MAX(ORDERDATE) FROM Sales.SalesOrderHeader))
    EXEC Sales.GetNonNullRecentSales @date;
END
GO
IF OBJECT_ID ( 'Sales.GetNonNullRecentSales', 'P') IS NOT NULL
    DROP PROCEDURE Sales.GetNonNullRecentSales;
GO
CREATE PROCEDURE Sales.GetNonNullRecentSales (@date datetime)
AS BEGIN
    SELECT * FROM Sales.SalesOrderHeader h, Sales.SalesOrderDetail d
    WHERE h.SalesOrderID = d.SalesOrderID
    AND h.OrderDate > @date
END
GO
```

#### 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. For more information, see Query Hints (Transact-SQL).

For some applications, recompiling the query each time it executes might take too much time. The `OPTIMIZE FOR` query hint can help even if you don't use the `RECOMPILE` option. For example, you could add an `OPTIMIZE FOR` option to the stored procedure Sales.GetRecentSales to specify a specific date. The following example adds the `OPTIMIZE FOR` option to the Sales.GetRecentSales procedure.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID ( 'Sales.GetRecentSales', 'P') IS NOT NULL
    DROP PROCEDURE Sales.GetRecentSales;
GO
CREATE PROCEDURE Sales.GetRecentSales (@date datetime)
AS BEGIN
    IF @date is NULL
        SET @date = DATEADD(MONTH, -3, (SELECT MAX(ORDERDATE) FROM Sales.SalesOrderHeader))
    SELECT * FROM Sales.SalesOrderHeader h, Sales.SalesOrderDetail d
    WHERE h.SalesOrderID = d.SalesOrderID
    AND h.OrderDate > @date
    OPTION ( OPTIMIZE FOR ( @date = '2004-05-01 00:00:00.000'))
END;
GO
```

#### Improving cardinality estimates with Plan Guides

For some applications, query design guidelines might not apply because you cannot change the query or using the RECOMPILE query hint might be cause too many recompiles. You can use plan guides to specify other hints, such as USE PLAN, to control the behavior of the query while investigating application changes with the application vendor. For more information about plan guides, see Plan Guides.

## See Also

CREATE STATISTICS (Transact-SQL)
UPDATE STATISTICS (Transact-SQL)
sp_updatestats (Transact-SQL)

DBCC SHOW_STATISTICS (Transact-SQL)
ALTER DATABASE SET Options (Transact-SQL)
DROP STATISTICS (Transact-SQL)
CREATE INDEX (Transact-SQL)
ALTER INDEX (Transact-SQL)
Create Filtered Indexes
Controlling Autostat (AUTO_UPDATE_STATISTICS) behavior in SQL Server
STATS_DATE (Transact-SQL)
sys.dm_db_stats_properties (Transact-SQL)
sys.dm_db_stats_histogram (Transact-SQL)
sys.stats
sys.stats_columns (Transact-SQL)

# Stored Procedures (Database Engine)

5/3/2018 • 5 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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. Procedures resemble constructs in other programming languages because they can:

- 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

The following list describes some benefits of using 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. Without the code encapsulation provided by a procedure, every individual line of code would have to cross 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 procedure controls what processes and activities are performed and protects the underlying database objects. This eliminates the requirement to grant permissions at the individual object level and simplifies the security layers.

The EXECUTE AS 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.

Using procedure parameters helps guard against SQL injection attacks. Since parameter input is treated as a literal value and not as executable code, it is more difficult for an attacker to insert a command into the Transact-SQL statement(s) inside the procedure and compromise security.

Procedures can be encrypted, helping to obfuscate the source code. For more information, see SQL Server Encryption.

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.

If there has been significant change to the tables or data referenced by the procedure, the precompiled plan may actually cause the procedure to perform slower. In this case, recompiling the procedure and forcing a new execution plan can improve performance.

## 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. For a complete list of system procedures, see System Stored Procedures (Transact-SQL)

SQL Server supports the system procedures that provide an interface from SQL Server to external programs for various maintenance activities. These extended procedures use the xp_ prefix. For a complete list of extended procedures, see General Extended Stored Procedures (Transact-SQL).

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.

## Related Tasks

| Task Description | Topic |
| --- | --- |
| Describes how to create a stored procedure. | Create a Stored Procedure |
| Describes how to modify a stored procedure. | Modify a Stored Procedure |
| Describes how to delete a stored procedure. | Delete a Stored Procedure |
| Describes how to execute a stored procedure. | Execute a Stored Procedure |
| Describes how to grant permissions on a stored procedure. | Grant Permissions on a Stored Procedure |
| Describes how to return data from a stored procedure to an application. | Return Data from a Stored Procedure |
| Describes how to recompile a stored procedure. | Recompile a Stored Procedure |
| Describes how to rename a stored procedure. | Rename a Stored Procedure |
| Describes how to view the definition of a stored procedure. | View the Definition of a Stored Procedure |
| Describes how to view the dependencies on a stored procedure. | View the Dependencies of a Stored Procedure |
| Describes how Parameters are used in a stored procedure. | Parameters |

## Related Content

CLR Stored Procedures

# Subqueries (SQL Server)

5/3/2018 • 21 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

A subquery is a query that is nested inside a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement, or inside another subquery. A subquery can be used anywhere an expression is allowed. In this example a subquery is used as a column expression named MaxUnitPrice in a SELECT statement.

```
USE AdventureWorks2016;
GO
SELECT Ord.SalesOrderID, Ord.OrderDate,
    (SELECT MAX(OrdDet.UnitPrice)
     FROM Sales.SalesOrderDetail AS OrdDet
     WHERE Ord.SalesOrderID = OrdDet.SalesOrderID) AS MaxUnitPrice
FROM Sales.SalesOrderHeader AS Ord;
GO
```

## Subquery Fundamentals

A subquery is also called an inner query or inner select, while the statement containing a subquery is also called an outer query or outer select.

Many Transact-SQL statements that include subqueries can be alternatively formulated as joins. Other questions can be posed only with subqueries. In Transact-SQL, there is usually no performance difference between a statement that includes a subquery and a semantically equivalent version that does not. However, in some cases where existence must be checked, a join yields better performance. Otherwise, the nested query must be processed for each result of the outer query to ensure elimination of duplicates. In such cases, a join approach would yield better results. The following is an example showing both a subquery `SELECT` and a join `SELECT` that return the same result set:

```
USE AdventureWorks2016;
GO

/* SELECT statement built using a subquery. */
SELECT Name
FROM Production.Product
WHERE ListPrice =
    (SELECT ListPrice
     FROM Production.Product
     WHERE Name = 'Chainring Bolts' );
GO

/* SELECT statement built using a join that returns
   the same result set. */
SELECT Prd1. Name
FROM Production.Product AS Prd1
     JOIN Production.Product AS Prd2
       ON (Prd1.ListPrice = Prd2.ListPrice)
WHERE Prd2. Name = 'Chainring Bolts';
GO
```

A subquery nested in the outer SELECT statement has the following components:

- A regular `SELECT` query including the regular select list components.
- A regular `FROM` clause including one or more table or view names.
- An optional `WHERE` clause.
- An optional `GROUP BY` clause.
- An optional `HAVING` clause.

The SELECT query of a subquery is always enclosed in parentheses. It cannot include a `COMPUTE` or `FOR BROWSE` clause, and may only include an `ORDER BY` clause when a TOP clause is also specified.

A subquery can be nested inside the `WHERE` or `HAVING` clause of an outer `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement, or inside another subquery. Up to 32 levels of nesting is possible, although the limit varies based on available memory and the complexity of other expressions in the query. Individual queries may not support nesting up to 32 levels. A subquery can appear anywhere an expression can be used, if it returns a single value.

If a table appears only in a subquery and not in the outer query, then columns from that table cannot be included in the output (the select list of the outer query).

Statements that include a subquery usually take one of these formats:

- WHERE expression [NOT] IN (subquery)
- WHERE expression comparison_operator [ANY | ALL] (subquery)
- WHERE [NOT] EXISTS (subquery)

In some Transact-SQL statements, the subquery can be evaluated as if it were an independent query. Conceptually, the subquery results are substituted into the outer query (although this is not necessarily how SQL Server actually processes Transact-SQL statements with subqueries).

There are three basic types of subqueries. Those that:

- Operate on lists introduced with `IN`, or those that a comparison operator modified by `ANY` or `ALL`.
- Are introduced with an unmodified comparison operator and must return a single value.
- Are existence tests introduced with `EXISTS`.

## Subquery rules

A subquery is subject to the following restrictions:

- The select list of a subquery introduced with a comparison operator can include only one expression or column name (except that `EXISTS` and `IN` operate on `SELECT *` or a list, respectively).
- If the `WHERE` clause of an outer query includes a column name, it must be join-compatible with the column in the subquery select list.
- The **ntext**, **text**, and **image** data types cannot be used in the select list of subqueries.
- Because they must return a single value, subqueries introduced by an unmodified comparison operator (one not followed by the keyword ANY or ALL) cannot include `GROUP BY` and `HAVING` clauses.
- The `DISTINCT` keyword cannot be used with subqueries that include GROUP BY.
- The `COMPUTE` and `INTO` clauses cannot be specified.
- `ORDER BY` can only be specified when `TOP` is also specified.
- A view created by using a subquery cannot be updated.
- The select list of a subquery introduced with `EXISTS`, by convention, has an asterisk (*) instead of a single column name. The rules for a subquery introduced with `EXISTS` are the same as those for a standard select list, because a subquery introduced with `EXISTS` creates an existence test and returns TRUE or FALSE, instead of data.

## Qualifying column names in subqueries

In the following example, the *CustomerID* column in the `WHERE` clause of the outer query is implicitly qualified by the table name in the outer query `FROM` clause (*Sales.Store*). The reference to *CustomerID* in the select list of the subquery is qualified by the subquery `FROM` clause, that is, by the *Sales.Customer* table.

```
USE AdventureWorks2016;
GO
SELECT Name
FROM Sales.Store
WHERE BusinessEntityID NOT IN
    (SELECT CustomerID
     FROM Sales.Customer
     WHERE TerritoryID = 5);
GO
```

The general rule is that column names in a statement are implicitly qualified by the table referenced in the `FROM` clause at the same level. If a column does not exist in the table referenced in the `FROM` clause of a subquery, it is implicitly qualified by the table referenced in the `FROM` clause of the outer query.

Here is what the query looks like with these implicit assumptions specified:

```
USE AdventureWorks2016;
GO
SELECT Name
FROM Sales.Store
WHERE Sales.Store.BusinessEntityID NOT IN
    (SELECT Sales.Customer.CustomerID
     FROM Sales.Customer
     WHERE TerritoryID = 5);
GO
```

It is never wrong to state the table name explicitly, and it is always possible to override implicit assumptions about table names with explicit qualifications.

**IMPORTANT**

If a column is referenced in a subquery that does not exist in the table referenced by the subquery's `FROM` clause, but exists in a table referenced by the outer query's `FROM` clause, the query executes without error. SQL Server implicitly qualifies the column in the subquery with the table name in the outer query.

## Multiple levels of nesting

A subquery can itself include one or more subqueries. Any number of subqueries can be nested in a statement.

The following query finds the names of employees who are also sales persons.

```
USE AdventureWorks2016;
GO
SELECT LastName, FirstName
FROM Person.Person
WHERE BusinessEntityID IN
    (SELECT BusinessEntityID
     FROM HumanResources.Employee
     WHERE BusinessEntityID IN
        (SELECT BusinessEntityID
         FROM Sales.SalesPerson)
    );
GO
```

Here is the result set.

```
LastName                                           FirstName
-------------------------------------------------- -----------------------
Jiang                                              Stephen
Abbas                                              Syed
Alberts                                            Amy
Ansman-Wolfe                                       Pamela
Campbell                                           David
Carson                                             Jillian
Ito                                                Shu
Mitchell                                           Linda
Reiter                                             Tsvi
Saraiva                                            Jos
Vargas                                             Garrett
Varkey Chudukatil                                  Ranjit
Valdez                                             Rachel
Tsoflias                                           Lynn
Pak                                                Jae
Blythe                                             Michael
Mensa-Annan                                        Tete

(17 row(s) affected)
```

The innermost query returns the sales person IDs. The query at the next higher level is evaluated with these sales person IDs and returns the contact ID numbers of the employees. Finally, the outer query uses the contact IDs to find the names of the employees.

You can also express this query as a join:

```
USE AdventureWorks2016;
GO
SELECT LastName, FirstName
FROM Person.Person c
INNER JOIN HumanResources.Employee e
ON c.BusinessEntityID = e.BusinessEntityID
JOIN Sales.SalesPerson s
ON e.BusinessEntityID = s.BusinessEntityID;
GO
```

# Correlated subqueries

Many queries can be evaluated by executing the subquery once and substituting the resulting value or values into the `WHERE` clause of the outer query. In queries that include a correlated subquery (also known as a repeating subquery), the subquery depends on the outer query for its values. This means that the subquery is executed repeatedly, once for each row that might be selected by the outer query. This query retrieves one instance of each employee's first and last name for which the bonus in the *SalesPerson* table is 5000 and for which the employee

identification numbers match in the *Employee* and *SalesPerson* tables.

```
USE AdventureWorks2016;
GO
SELECT DISTINCT c.LastName, c.FirstName, e.BusinessEntityID
FROM Person.Person AS c JOIN HumanResources.Employee AS e
ON e.BusinessEntityID = c.BusinessEntityID
WHERE 5000.00 IN
    (SELECT Bonus
    FROM Sales.SalesPerson sp
    WHERE e.BusinessEntityID = sp.BusinessEntityID) ;
GO
```

Here is the result set.

```
LastName FirstName BusinessEntityID
-------------------------- ---------- ------------
Ansman-Wolfe Pamela 280
Saraiva José 282

(2 row(s) affected)
```

The previous subquery in this statement cannot be evaluated independently of the outer query. It needs a value for *Employee.BusinessEntityID*, but this value changes as SQL Server examines different rows in *Employee*. That is exactly how this query is evaluated: SQL Server considers each row of the Employee table for inclusion in the results by substituting the value in each row into the inner query. For example, if SQL Server first examines the row for `Syed Abbas`, the variable *Employee.BusinessEntityID* takes the value 285, which SQL Server substitutes into the inner query.

```
USE AdventureWorks2016;
GO
SELECT Bonus
FROM Sales.SalesPerson
WHERE BusinessEntityID = 285;
GO
```

The result is 0 (`Syed Abbas` did not receive a bonus because he is not a sales person), so the outer query evaluates to:

```
USE AdventureWorks2016;
GO
SELECT LastName, FirstName
FROM Person.Person AS c JOIN HumanResources.Employee AS e
ON e.BusinessEntityID = c.BusinessEntityID
WHERE 5000 IN (0.00);
GO
```

Because this is false, the row for `Syed Abbas` is not included in the results. Go through the same procedure with the row for `Pamela Ansman-Wolfe`. You will see that this row is included in the results.

Correlated subqueries can also include table-valued functions in the `FROM` clause by referencing columns from a table in the outer query as an argument of the table-valued function. In this case, for each row of the outer query, the table-valued function is evaluated according to the subquery.

## Subquery types

Subqueries can be specified in many places:

- With aliases. For more information, see Subqueries with Aliases.
- With `IN` or `NOT IN`. For more information, see Subqueries with IN and Subqueries with NOT IN.
- In `UPDATE`, `DELETE`, and `INSERT` statements. For more information, see Subqueries in UPDATE, DELETE, and INSERT Statements.
- With comparison operators. For more information, see Subqueries with Comparison Operators.
- With `ANY`, `SOME`, or `ALL`. For more information, see Comparison Operators Modified by ANY, SOME, or ALL.
- With `EXISTS` or `NOT EXISTS`. For more information, see Subqueries with EXISTS and Subqueries with NOT EXISTS.
- In place of an expression. For more information, see Subqueries used in place of an Expression.

#### Subqueries with Aliases

Many statements in which the subquery and the outer query refer to the same table can be stated as self-joins (joining a table to itself). For example, you can find addresses of employees from a particular state using a subquery:

```
USE AdventureWorks2016;
GO
SELECT StateProvinceID, AddressID
FROM Person.Address
WHERE AddressID IN
    (SELECT AddressID
     FROM Person.Address
     WHERE StateProvinceID = 39);
GO
```

Here is the result set.

```
StateProvinceID AddressID
----------- -----------
39 942
39 955
39 972
39 22660

(4 row(s) affected)
```

Or you can use a self-join:

```
USE AdventureWorks2016;
GO
SELECT e1.StateProvinceID, e1.AddressID
FROM Person.Address AS e1
INNER JOIN Person.Address AS e2
ON e1.AddressID = e2.AddressID
AND e2.StateProvinceID = 39;
GO
```

Table aliases are required because the table being joined to itself appears in two different roles. Aliases can also be used in nested queries that refer to the same table in an inner and outer query.

```
USE AdventureWorks2016;
GO
SELECT e1.StateProvinceID, e1.AddressID
FROM Person.Address AS e1
WHERE e1.AddressID IN
    (SELECT e2.AddressID
     FROM Person.Address AS e2
     WHERE e2.StateProvinceID = 39);
GO
```

Explicit aliases make it clear that a reference to *Person.Address* in the subquery does not mean the same thing as the reference in the outer query.

### Subqueries with IN

The result of a subquery introduced with `IN` (or with `NOT IN`) is a list of zero or more values. After the subquery returns results, the outer query makes use of them.
The following query finds the names of all the wheel products that Adventure Works Cycles makes.

```
USE AdventureWorks2016;
GO
SELECT Name
FROM Production.Product
WHERE ProductSubcategoryID IN
    (SELECT ProductSubcategoryID
     FROM Production.ProductSubcategory
     WHERE Name = 'Wheels');
GO
```

Here is the result set.

```
Name
----------------------------
LL Mountain Front Wheel
ML Mountain Front Wheel
HL Mountain Front Wheel
LL Road Front Wheel
ML Road Front Wheel
HL Road Front Wheel
Touring Front Wheel
LL Mountain Rear Wheel
ML Mountain Rear Wheel
HL Mountain Rear Wheel
LL Road Rear Wheel
ML Road Rear Wheel
HL Road Rear Wheel
Touring Rear Wheel

(14 row(s) affected)
```

This statement is evaluated in two steps. First, the inner query returns the subcategory identification number that matches the name 'Wheel' (17). Second, this value is substituted into the outer query, which finds the product names that go with the subcategory identification numbers in Product.

```
USE AdventureWorks2016;
GO
SELECT Name
FROM Production.Product
WHERE ProductSubcategoryID IN ('17');
GO
```

One difference in using a join rather than a subquery for this and similar problems is that the join lets you show columns from more than one table in the result. For example, if you want to include the name of the product subcategory in the result, you must use a join version.

```
USE AdventureWorks2016;
GO
SELECT p.Name, s.Name
FROM Production.Product p
INNER JOIN Production.ProductSubcategory s
ON p.ProductSubcategoryID = s.ProductSubcategoryID
AND s.Name = 'Wheels';
GO
```

Here is the result set.

```
Name
LL Mountain Front Wheel Wheels
ML Mountain Front Wheel Wheels
HL Mountain Front Wheel Wheels
LL Road Front Wheel Wheels
ML Road Front Wheel Wheels
HL Road Front Wheel Wheels
Touring Front Wheel Wheels
LL Mountain Rear Wheel Wheels
ML Mountain Rear Wheel Wheels
HL Mountain Rear Wheel Wheels
LL Road Rear Wheel Wheels
ML Road Rear Wheel Wheels
HL Road Rear Wheel Wheels
Touring Rear Wheel Wheels

(14 row(s) affected)
```

The following query finds the name of all vendors whose credit rating is good, from whom Adventure Works Cycles orders at least 20 items, and whose average lead time to deliver is less than 16 days.

```
USE AdventureWorks2016;
GO
SELECT Name
FROM Purchasing.Vendor
WHERE CreditRating = 1
AND BusinessEntityID IN
    (SELECT BusinessEntityID
     FROM Purchasing.ProductVendor
     WHERE MinOrderQty >= 20
     AND AverageLeadTime < 16);
GO
```

Here is the result set.

```
Name
--------------------------------------------------
Compete Enterprises, Inc
International Trek Center
First National Sport Co.
Comfort Road Bicycles
Circuit Cycles
First Rate Bicycles
Jeff's Sporting Goods
Competition Bike Training Systems
Electronic Bike Repair & Supplies
Crowley Sport
Expert Bike Co
Team Athletic Co.
Compete, Inc.

(13 row(s) affected)
```

The inner query is evaluated, producing the ID numbers of the vendors who meet the subquery qualifications. The outer query is then evaluated. Notice that you can include more than one condition in the WHERE clause of both the inner and the outer query.

Using a join, the same query is expressed like this:

```
USE AdventureWorks2016;
GO
SELECT DISTINCT Name
FROM Purchasing.Vendor v
INNER JOIN Purchasing.ProductVendor p
ON v.BusinessEntityID = p.BusinessEntityID
WHERE CreditRating = 1
  AND MinOrderQty >= 20
  AND AverageLeadTime < 16;
GO
```

A join can always be expressed as a subquery. A subquery can often, but not always, be expressed as a join. This is because joins are symmetric: you can join table A to B in either order and get the same answer. The same is not true if a subquery is involved.

**Subqueries with NOT IN**

Subqueries introduced with the keyword NOT IN also return a list of zero or more values.
The following query finds the names of the products that are not finished bicycles.

```
USE AdventureWorks2016;
GO
SELECT Name
FROM Production.Product
WHERE ProductSubcategoryID NOT IN
    (SELECT ProductSubcategoryID
     FROM Production.ProductSubcategory
     WHERE Name = 'Mountain Bikes'
        OR Name = 'Road Bikes'
        OR Name = 'Touring Bikes');
GO
```

This statement cannot be converted to a join. The analogous not-equal join has a different meaning: It finds the names of products that are in some subcategory that is not a finished bicycle.

**Subqueries in UPDATE, DELETE, and INSERT Statements**

Subqueries can be nested in the `UPDATE`, `DELETE`, `INSERT` and `SELECT` data manipulation (DML) statements.

The following example doubles the value in the *ListPrice* column in the *Production.Product* table. The subquery in the `WHERE` clause references the *Purchasing.ProductVendor* table to restrict the rows updated in the *Product* table to just those supplied by *BusinessEntity* 1540.

```
USE AdventureWorks2016;
GO
UPDATE Production.Product
SET ListPrice = ListPrice * 2
WHERE ProductID IN
    (SELECT ProductID
     FROM Purchasing.ProductVendor
     WHERE BusinessEntityID = 1540);
GO
```

Here is an equivalent `UPDATE` statement using a join:

```
USE AdventureWorks2016;
GO
UPDATE Production.Product
SET ListPrice = ListPrice * 2
FROM Production.Product AS p
INNER JOIN Purchasing.ProductVendor AS pv
    ON p.ProductID = pv.ProductID AND BusinessEntityID = 1540;
GO
```

**Subqueries with Comparison Operators**

Subqueries can be introduced with one of the comparison operators (=, < >, >, > =, <, ! >, ! <, or < =).

A subquery introduced with an unmodified comparison operator (a comparison operator not followed by `ANY` or `ALL`) must return a single value rather than a list of values, like subqueries introduced with `IN`. If such a subquery returns more than one value, SQL Server displays an error message.

To use a subquery introduced with an unmodified comparison operator, you must be familiar enough with your data and with the nature of the problem to know that the subquery will return exactly one value.

For example, if you assume each sales person only covers one sales territory, and you want to find the customers located in the territory covered by `Linda Mitchell`, you can write a statement with a subquery introduced with the simple = comparison operator.

```
USE AdventureWorks2016;
GO
SELECT CustomerID
FROM Sales.Customer
WHERE TerritoryID =
    (SELECT TerritoryID
     FROM Sales.SalesPerson
     WHERE BusinessEntityID = 276);
GO
```

If, however, `Linda Mitchell` covered more than one sales territory, then an error message would result. Instead of the = comparison operator, an `IN` formulation could be used (= ANY also works).

Subqueries introduced with unmodified comparison operators often include aggregate functions, because these return a single value. For example, the following statement finds the names of all products whose list price is greater than the average list price.

```
USE AdventureWorks2016;
GO
SELECT Name
FROM Production.Product
WHERE ListPrice >
    (SELECT AVG (ListPrice)
     FROM Production.Product);
GO
```

Because subqueries introduced with unmodified comparison operators must return a single value, they cannot include `GROUP BY` or `HAVING` clauses unless you know the GROUP BY or HAVING clause itself returns a single value. For example, the following query finds the products priced higher than the lowest-priced product that is in subcategory 14.

```
USE AdventureWorks2016;
GO
SELECT Name
FROM Production.Product
WHERE ListPrice >
    (SELECT MIN (ListPrice)
     FROM Production.Product
     GROUP BY ProductSubcategoryID
     HAVING ProductSubcategoryID = 14);
GO
```

**Comparison Operators Modified by ANY, SOME, or ALL**

Comparison operators that introduce a subquery can be modified by the keywords ALL or ANY. SOME is an ISO standard equivalent for `ANY`.

Subqueries introduced with a modified comparison operator return a list of zero or more values and can include a `GROUP BY` or `HAVING` clause. These subqueries can be restated with `EXISTS`.

Using the > comparison operator as an example, `>ALL` means greater than every value. In other words, it means greater than the maximum value. For example, `>ALL (1, 2, 3)` means greater than 3. `>ANY` means greater than at least one value, that is, greater than the minimum. So `>ANY (1, 2, 3)` means greater than 1. For a row in a subquery with `>ALL` to satisfy the condition specified in the outer query, the value in the column introducing the subquery must be greater than each value in the list of values returned by the subquery.

Similarly, `>ANY` means that for a row to satisfy the condition specified in the outer query, the value in the column that introduces the subquery must be greater than at least one of the values in the list of values returned by the subquery.

The following query provides an example of a subquery introduced with a comparison operator modified by ANY. It finds the products whose list prices are greater than or equal to the maximum list price of any product subcategory.

```
USE AdventureWorks2016;
GO
SELECT Name
FROM Production.Product
WHERE ListPrice >= ANY
    (SELECT MAX (ListPrice)
     FROM Production.Product
     GROUP BY ProductSubcategoryID);
GO
```

For each Product subcategory, the inner query finds the maximum list price. The outer query looks at all of these

values and determines which individual product's list prices are greater than or equal to any product subcategory's maximum list price. If `ANY` is changed to `ALL`, the query will return only those products whose list price is greater than or equal to all the list prices returned in the inner query.

If the subquery does not return any values, the entire query fails to return any values.

The `=ANY` operator is equivalent to `IN`. For example, to find the names of all the wheel products that Adventure Works Cycles makes, you can use either `IN` or `=ANY`.

```
--Using =ANY
USE AdventureWorks2016;
GO
SELECT Name
FROM Production.Product
WHERE ProductSubcategoryID =ANY
    (SELECT ProductSubcategoryID
     FROM Production.ProductSubcategory
     WHERE Name = 'Wheels');
GO

--Using IN
USE AdventureWorks2016;
GO
SELECT Name
FROM Production.Product
WHERE ProductSubcategoryID IN
    (SELECT ProductSubcategoryID
     FROM Production.ProductSubcategory
     WHERE Name = 'Wheels');
GO
```

Here is the result set for either query:

```
Name
--------------------------------------------------
LL Mountain Front Wheel
ML Mountain Front Wheel
HL Mountain Front Wheel
LL Road Front Wheel
ML Road Front Wheel
HL Road Front Wheel
Touring Front Wheel
LL Mountain Rear Wheel
ML Mountain Rear Wheel
HL Mountain Rear Wheel
LL Road Rear Wheel
ML Road Rear Wheel
HL Road Rear Wheel
Touring Rear Wheel

(14 row(s) affected)
```

The `<>ANY` operator, however, differs from `NOT IN`: `<>ANY` means not = a, or not = b, or not = c. `NOT IN` means not = a, and not = b, and not = c. `<>ALL` means the same as `NOT IN`.

For example, the following query finds customers located in a territory not covered by any sales persons.

```
USE AdventureWorks2016;
GO
SELECT CustomerID
FROM Sales.Customer
WHERE TerritoryID <> ANY
    (SELECT TerritoryID
     FROM Sales.SalesPerson);
GO
```

The results include all customers, except those whose sales territories are NULL, because every territory that is assigned to a customer is covered by a sales person. The inner query finds all the sales territories covered by sales persons, and then, for each territory, the outer query finds the customers who are not in one.

For the same reason, when you use `NOT IN` in this query, the results include none of the customers.

You can get the same results with the `<>ALL` operator, which is equivalent to `NOT IN`.

**Subqueries with EXISTS**

When a subquery is introduced with the keyword `EXISTS`, the subquery functions as an existence test. The `WHERE` clause of the outer query tests whether the rows that are returned by the subquery exist. The subquery does not actually produce any data; it returns a value of TRUE or FALSE.

A subquery introduced with EXISTS has the following syntax:

`WHERE [NOT] EXISTS (subquery)`

The following query finds the names of all products that are in the Wheels subcategory:

```
USE AdventureWorks2016;
GO
SELECT Name
FROM Production.Product
WHERE EXISTS
    (SELECT *
     FROM Production.ProductSubcategory
     WHERE ProductSubcategoryID =
            Production.Product.ProductSubcategoryID
        AND Name = 'Wheels');
GO
```

Here is the result set.

```
Name
--------------------------------------------------
LL Mountain Front Wheel
ML Mountain Front Wheel
HL Mountain Front Wheel
LL Road Front Wheel
ML Road Front Wheel
HL Road Front Wheel
Touring Front Wheel
LL Mountain Rear Wheel
ML Mountain Rear Wheel
HL Mountain Rear Wheel
LL Road Rear Wheel
ML Road Rear Wheel
HL Road Rear Wheel
Touring Rear Wheel

(14 row(s) affected)
```

To understand the results of this query, consider the name of each product in turn. Does this value cause the subquery to return at least one row? In other words, does the query cause the existence test to evaluate to TRUE?

Notice that subqueries that are introduced with EXISTS are a bit different from other subqueries in the following ways:

- The keyword `EXISTS` is not preceded by a column name, constant, or other expression.
- The select list of a subquery introduced by `EXISTS` almost always consists of an asterisk (*). There is no reason to list column names because you are just testing whether rows that meet the conditions specified in the subquery exist.

The `EXISTS` keyword is important because frequently there is no alternative formulation without subqueries. Although some queries that are created with EXISTS cannot be expressed any other way, many queries can use IN or a comparison operator modified by `ANY` or `ALL` to achieve similar results.

For example, the preceding query can be expressed by using IN:

```
USE AdventureWorks2016;
GO
SELECT Name
FROM Production.Product
WHERE ProductSubcategoryID IN
    (SELECT ProductSubcategoryID
     FROM Production.ProductSubcategory
     WHERE Name = 'Wheels');
GO
```

### Subqueries with NOT EXISTS

`NOT EXISTS` works like `EXISTS`, except the `WHERE` clause in which it is used is satisfied if no rows are returned by the subquery.

For example, to find the names of products that are not in the wheels subcategory:

```
USE AdventureWorks2016;
GO
SELECT Name
FROM Production.Product
WHERE NOT EXISTS
    (SELECT *
     FROM Production.ProductSubcategory
     WHERE ProductSubcategoryID =
            Production.Product.ProductSubcategoryID
        AND Name = 'Wheels');
GO
```

### Subqueries Used in place of an Expression

In Transact-SQL, a subquery can be substituted anywhere an expression can be used in `SELECT`, `UPDATE`, `INSERT`, and `DELETE` statements, except in an `ORDER BY` list.

The following example illustrates how you might use this enhancement. This query finds the prices of all mountain bike products, their average price, and the difference between the price of each mountain bike and the average price.

```
USE AdventureWorks2016;
GO
SELECT Name, ListPrice,
(SELECT AVG(ListPrice) FROM Production.Product) AS Average,
    ListPrice - (SELECT AVG(ListPrice) FROM Production.Product)
    AS Difference
FROM Production.Product
WHERE ProductSubcategoryID = 1;
GO
```

## See Also

IN (Transact-SQL)
EXISTS (Transact-SQL)
ALL (Transact-SQL)
SOME | ANY (Transact-SQL)
Joins
Comparison Operators (Transact-SQL)

# Synonyms (Database Engine)

5/3/2018 • 5 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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**.

  A synonym belongs to a schema, and like other objects in a schema, the name of a synonym must be unique. You can create synonyms for the following database objects:

| | |
|---|---|
| Assembly (CLR) stored procedure | Assembly (CLR) table-valued function |
| Assembly (CLR) scalar function | Assembly (CLR) aggregate functions |
| Replication-filter-procedure | Extended stored procedure |
| SQL scalar function | SQL table-valued function |
| SQL inline-tabled-valued function | SQL stored procedure |
| View | Table* (User-defined) |

*Includes local and global temporary tables

> **NOTE**
>
> Four-part names for function base objects are not supported.

A synonym cannot be the base object for another synonym, and a synonym cannot reference a user-defined aggregate function.

The binding between a synonym and its base object is by name only. All existence, type, and permissions checking

on the base object is deferred until run time. Therefore, the base object can be modified, dropped, or dropped and replaced by another object that has the same name as the original base object. For example, consider a synonym, **MyContacts**, that references the **Person.Contact** table in Adventure Works. If the **Contact** table is dropped and replaced by a view named **Person.Contact**, **MyContacts** now references the **Person.Contact** view.

References to synonyms are not schema-bound. Therefore, a synonym can be dropped at any time. However, by dropping a synonym, you run the risk of leaving dangling references to the synonym that was dropped. These references will only be found at run time.

## Synonyms and Schemas

If you have a default schema that you do not own and want to create a synonym, you must qualify the synonym name with the name of a schema that you do own. For example, if you own a schema **x**, but **y** is your default schema and you use the CREATE SYNONYM statement, you must prefix the name of the synonym with the schema **x**, instead of naming the synonym by using a single-part name. For more information about how to create synonyms, see CREATE SYNONYM (Transact-SQL).

## Granting Permissions on a Synonym

Only synonym owners, members of **db_owner**, or members of **db_ddladmin** can grant permission on a synonym.

You can GRANT, DENY, REVOKE all or any of the following permissions on a synonym:

| | |
|---|---|
| CONTROL | DELETE |
| EXECUTE | INSERT |
| SELECT | TAKE OWNERSHIP |
| UPDATE | VIEW DEFINITION |

## Using Synonyms

You can use synonyms in place of their referenced base object in several SQL statements and expression contexts. The following table contains a list of these statements and expression contexts:

| | |
|---|---|
| SELECT | INSERT |
| UPDATE | DELETE |
| EXECUTE | Sub-selects |

When you are working with synonyms in the contexts previously stated, the base object is affected. For example, if a synonym references a base object that is a table and you insert a row into the synonym, you are actually inserting a row into the referenced table.

**NOTE**

You cannot reference a synonym that is located on a linked server.

You can use a synonym as the parameter for the OBJECT_ID function; however, the function returns the object ID

of the synonym, not the base object.

You cannot reference a synonym in a DDL statement. For example, the following statements, which reference a synonym named `dbo.MyProduct`, generate errors:

```
ALTER TABLE dbo.MyProduct
   ADD NewFlag int null;
EXEC ('ALTER TABLE dbo.MyProduct
   ADD NewFlag int null');
```

The following permission statements are associated only with the synonym and not the base object:

| | |
|---|---|
| GRANT | DENY |
| REVOKE | |

Synonyms are not schema-bound and, therefore, cannot be referenced by the following schema-bound expression contexts:

| | |
|---|---|
| CHECK constraints | Computed columns |
| Default expressions | Rule expressions |
| Schema-bound views | Schema-bound functions |

For more information about schema-bound functions, see Create User-defined Functions (Database Engine).

# Getting Information About Synonyms

The sys.synonyms catalog view contains an entry for each synonym in a given database. This catalog view exposes synonym metadata such as the name of the synonym and the name of the base object. For more information about the **sys.synonyms** catalog view, see sys.synonyms (Transact-SQL).

By using extended properties, you can add descriptive or instructional text, input masks, and formatting rules as properties of a synonym. Because the property is stored in the database, all applications that read the property can evaluate the object in the same way. For more information, see sp_addextendedproperty (Transact-SQL).

To find the base type of the base object of a synonym, use the OBJECTPROPERTYEX function. For more information, see OBJECTPROPERTYEX (Transact-SQL).

### Examples

The following example returns the base type of a synonym's base object that is a local object.

```
USE tempdb;
GO
CREATE SYNONYM MyEmployee
FOR AdventureWorks2012.HumanResources.Employee;
GO
SELECT OBJECTPROPERTYEX(OBJECT_ID('MyEmployee'), 'BaseType') AS BaseType;
```

The following example returns the base type of a synonym's base object that is a remote object located on a server named `Server1`.

```
EXECUTE sp_addlinkedserver Server1;
GO
CREATE SYNONYM MyRemoteEmployee
FOR Server1.AdventureWorks2012.HumanResources.Employee;
GO
SELECT OBJECTPROPERTYEX(OBJECT_ID('MyRemoteEmployee'), 'BaseType') AS BaseType;
GO
```

## Related Content

Create Synonyms

CREATE SYNONYM (Transact-SQL)

DROP SYNONYM (Transact-SQL)

# System Catalog Views (Transact-SQL)

5/3/2018 • 2 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Catalog views return information that is used by the SQL Server Database Engine. We recommend that you use catalog views because they are the most general interface to the catalog metadata and provide the most efficient way to obtain, transform, and present customized forms of this information. 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 catalog view inherits from the sys.objects catalog view. The sys.objects catalog view is referred to as the base view, and the sys.tables view is called the derived view. The sys.tables catalog view returns the columns that are specific to tables and also all the columns that the sys.objects catalog view returns. The sys.objects catalog view returns rows for objects other than tables, such as stored procedures and views. After a table is created, the metadata for the table is returned in both views. Although the two catalog views return different levels of information about the table, there is only one entry in metadata for this table with one name and one object_id. This can be summarized as follows:

- The base view contains a subset of columns and a superset of rows.
- The derived view contains a superset of columns and a subset of rows.

> **IMPORTANT**
> In future releases of SQL Server, Microsoft may augment the definition of any system catalog view by adding columns to the end of the column list. We recommend against using the syntax SELECT * FROM *sys.catalog_view_name* in production code because the number of columns returned might change and break your application.

The catalog views in SQL Server have been organized into the following categories:

| | |
|---|---|
| Always On Availability Groups Catalog Views (Transact-SQL) | Messages (for Errors) Catalog Views (Transact-SQL) |
| Azure SQL Database Catalog Views | Object Catalog Views (Transact-SQL) |
| Change Tracking Catalog Views (Transact-SQL) | Partition Function Catalog Views (Transact-SQL) |
| CLR Assembly Catalog Views (Transact-SQL) | Policy-Based Management Views (Transact-SQL) |
| Data Collector Views (Transact-SQL) | Resource Governor Catalog Views (Transact-SQL) |
| Data Spaces (Transact-SQL) | Query Store Catalog Views (Transact-SQL) |

| | |
|---|---|
| Database Mail Views (Transact-SQL) | Scalar Types Catalog Views (Transact-SQL) |
| Database Mirroring Witness Catalog Views (Transact-SQL) | Schemas Catalog Views (Transact-SQL) |
| Databases and Files Catalog Views (Transact-SQL) | Security Catalog Views (Transact-SQL) |
| Endpoints Catalog Views (Transact-SQL) | Service Broker Catalog Views (Transact-SQL) |
| Extended Events Catalog Views (Transact-SQL) | Server-wide Configuration Catalog Views (Transact-SQL) |
| Extended Properties Catalog Views (Transact-SQL) | Spatial Data Catalog Views |
| External Operations Catalog Views (Transact-SQL) | SQL Data Warehouse and Parallel Data Warehouse Catalog Views |
| Filestream and FileTable Catalog Views (Transact-SQL) | Stretch Database Catalog Views (Transact-SQL) |
| Full-Text Search and Semantic Search Catalog Views (Transact-SQL) | XML Schemas (XML Type System) Catalog Views (Transact-SQL) |
| Linked Servers Catalog Views (Transact-SQL) | |

## See Also

Information Schema Views (Transact-SQL)
System Tables (Transact-SQL)
Querying the SQL Server System Catalog FAQ

# System Compatibility Views (Transact-SQL)

5/3/2018 • 2 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2012) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Many of the system tables from earlier releases of SQL Server are now implemented as a set of views. These views are known as compatibility views, and they are meant for backward compatibility only. The compatibility views expose the same metadata that was available in SQL Server 2000 (8.x). However, the compatibility views do not expose any of the metadata related to features that are introduced in SQL Server 2005 and later. Therefore, when you use new features, such as Service Broker or partitioning, you must switch to using the catalog views.

Another reason for upgrading to the catalog views is that compatibility view columns that store user IDs and type IDs may return NULL or trigger arithmetic overflows. This is because you can create more than 32,767 users, groups, and roles, and 32,767 data types. For example, if you were to create 32,768 users, and then run the following query: `SELECT * FROM sys.sysusers`. If ARITHABORT is set to ON, the query fails with an arithmetic overflow error. If ARITHABORT is set to OFF, the **uid** column returns NULL.

To avoid these problems, we recommend that you use the new catalog views that can handle the increased number of user IDs and type IDs. The following table lists the columns that are subject to this overflow.

| COLUMN NAME | COMPATIBILITY VIEW | SQL SERVER 2005 VIEW |
|---|---|---|
| **xusertype** | **syscolumns** | **sys.columns** |
| **usertype** | **syscolumns** | **sys.columns** |
| **memberuid** | **sysmembers** | **sys.database_role_members** |
| **groupuid** | **sysmembers** | **sys.database_role_members** |
| **uid** | **sysobjects** | **sys.objects** |
| **uid** | **sysprotects** | **sys.database_permissions**<br>**sys.server_permissions** |
| **grantor** | **sysprotects** | **sys.database_permissions**<br>**sys.server_permissions** |
| **xusertype** | **systypes** | **sys.types** |
| **uid** | **systypes** | **sys.types** |
| **uid** | **sysusers** | **sys.database_principals** |
| **altuid** | **sysusers** | **sys.database_principals** |
| **gid** | **sysusers** | **sys.database_principals** |

| COLUMN NAME | COMPATIBILITY VIEW | SQL SERVER 2005 VIEW |
| --- | --- | --- |
| **uid** | **syscacheobjects** | **sys.dm_exec_plan_attributes** |
| **uid** | **sysprocesses** | **sys.dm_exec_requests** |

When referenced in a user database, system tables which were announced as deprecated in SQL Server 2000 (such as **syslanguages** or **syscacheobjects**), are now bound to the back-compatibility view in the **sys** schema. Since the SQL Server 2000 system tables have been deprecated for multiple versions, this change is not considered a breaking change.

Example: If a user creates a user-table called **syslanguages** in a user-database, in SQL Server 2008, the statement `SELECT * from dbo.syslanguages;` in that database would return the values from the user table. Beginning in SQL Server 2012, this practice will return data from the system view **sys.syslanguages**.

## See Also

Catalog Views (Transact-SQL)
Mapping System Tables to System Views (Transact-SQL)

# System Dynamic Management Views

5/4/2018 • 3 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2008) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Dynamic management views and functions return server state information that can be used to monitor the health of a server instance, diagnose problems, and tune performance.

> **IMPORTANT**
>
> Dynamic management views and functions return internal, implementation-specific state data. Their schemas and the data they return may change in future releases of SQL Server. Therefore, dynamic management views and functions in future releases may not be compatible with the dynamic management views and functions in this release. For example, in future releases of SQL Server, Microsoft may augment the definition of any dynamic management view by adding columns to the end of the column list. We recommend against using the syntax `SELECT * FROM dynamic_management_view_name` in production code because the number of columns returned might change and break your application.

There are two types of dynamic management views and functions:

- Server-scoped dynamic management views and functions. These require VIEW SERVER STATE permission on the server.
- Database-scoped dynamic management views and functions. These require VIEW DATABASE STATE permission on the database.

## Querying Dynamic Management Views

Dynamic management views can be referenced in Transact-SQL statements by using two-part, three-part, or four-part names. Dynamic management functions on the other hand can be referenced in Transact-SQL statements by using either two-part or three-part names. Dynamic management views and functions cannot be referenced in Transact-SQL statements by using one-part names.

All dynamic management views and functions exist in the sys schema and follow this naming convention dm_*. When you use a dynamic management view or function, you must prefix the name of the view or function by using the sys schema. For example, to query the dm_os_wait_stats dynamic management view, run the following query:

```
SELECT wait_type, wait_time_ms
FROM sys.dm_os_wait_stats;
```

#### Required Permissions

To query a dynamic management view or function requires SELECT permission on object and VIEW SERVER STATE or VIEW DATABASE STATE permission. This lets you selectively restrict access of a user or login to dynamic management views and functions. To do this, first create the user in master and then deny the user SELECT permission on the dynamic management views or functions that you do not want them to access. After this, the user cannot select from these dynamic management views or functions, regardless of database context of the user.

**NOTE**

Because DENY takes precedence, if a user has been granted VIEW SERVER STATE permissions but denied VIEW DATABASE STATE permission, the user can see server-level information, but not database-level information.

## In This Section

Dynamic management views and functions have been organized into the following categories.

| | |
|---|---|
| Always On Availability Groups Dynamic Management Views and Funtions (Transact-SQL) | Memory-Optimized Table Dynamic Management Views (Transact-SQL) |
| Change Data Capture Related Dynamic Management Views (Transact-SQL) | Object Related Dynamic Management Views and Functions (Transact-SQL) |
| Change Tracking Related Dynamic Management Views | Query Notifications Related Dynamic Management Views (Transact-SQL) |
| Common Language Runtime Related Dynamic Management Views (Transact-SQL) | Replication Related Dynamic Management Views (Transact-SQL) |
| Database Mirroring Related Dynamic Management Views (Transact-SQL) | Resource Governor Related Dynamic Management Views (Transact-SQL) |
| Database Related Dynamic Management Views (Transact-SQL) | Security-Related Dynamic Management Views and Functions (Transact-SQL) |
| Execution Related Dynamic Management Views and Functions (Transact-SQL) | Server-Related Dynamic Management Views and Functions (Transact-SQL) |
| Extended Events Dynamic Management Views | Service Broker Related Dynamic Management Views (Transact-SQL) |
| Filestream and FileTable Dynamic Management Views (Transact-SQL) | Spatial Data Related Dynamic Management Views and Functions (Transact-SQL) |
| Full-Text Search and Semantic Search Dynamic Management Views and Functions (Transact-SQL) | SQL Data Warehouse and Parallel Data Warehouse Dynamic Management Views (Transact-SQL) |
| Geo-Replication Dynamic Management Views and Functions (Azure SQL Database) | SQL Server Operating System Related Dynamic Management Views (Transact-SQL) |
| Index Related Dynamic Management Views and Functions (Transact-SQL) | Stretch Database Dynamic Management Views (Transact-SQL) |
| I O Related Dynamic Management Views and Functions (Transact-SQL) | Transaction Related Dynamic Management Views and Functions (Transact-SQL) |

## See Also

GRANT Server Permissions (Transact-SQL)
GRANT Database Permissions (Transact-SQL)
System Views (Transact-SQL)

# System Functions for Transact-SQL

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2012) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

SQL Server provides the following groups of system functions.

## In This Section

Always On Availability Groups Functions
Change Data Capture Functions
Change Tracking Functions
Data Collector Functions
Filestream and FileTable Functions
Managed Backup Functions
sys.fn_get_sql
sys.fn_MSxe_read_event_stream
sys.fn_stmt_sql_handle_from_sql_stmt
sys.fn_validate_plan_guide
sys.fn_xe_file_target_read_file
sys.fn_backup_file_snapshots
Semantic Full-Text Search Functions
System Metadata Functions
System Security Functions
System Trace Functions

# System Information Schema Views (Transact-SQL)

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2008) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

An information schema view is one of several methods SQL Server provides for obtaining metadata. Information schema views provide an internal, system table-independent view of the SQL Server metadata. Information schema views enable applications to work correctly although significant changes have been made to the underlying system tables. The information schema views included in SQL Server comply with the ISO standard definition for the INFORMATION_SCHEMA.

> **IMPORTANT**
> Some changes have been made to the information schema views that break backward compatibility. These changes are described in the topics for the specific views.

SQL Server supports a three-part naming convention when you refer to the current server. The ISO standard also supports a three-part naming convention. However, the names used in both naming conventions are different. The information schema views are defined in a special schema named INFORMATION_SCHEMA. This schema is contained in each database. Each information schema view contains metadata for all data objects stored in that particular database. The following table shows the relationships between the SQL Server names and the SQL standard names.

| SQL SERVER NAME | MAPS TO THIS EQUIVALENT SQL STANDARD NAME |
| --- | --- |
| Database | Catalog |
| Schema | Schema |
| Object | Object |
| user-defined data type | Domain |

This name-mapping convention applies to the following SQL Server ISO-compatible views.

| | |
| --- | --- |
| CHECK_CONSTRAINTS | REFERENTIAL_CONSTRAINTS |
| COLUMN_DOMAIN_USAGE | ROUTINES |
| COLUMN_PRIVILEGES | ROUTINE_COLUMNS |
| COLUMNS | SCHEMATA |
| CONSTRAINT_COLUMN_USAGE | TABLE_CONSTRAINTS |
| CONSTRAINT_TABLE_USAGE | TABLE_PRIVILEGES |

| | |
|---|---|
| DOMAIN_CONSTRAINTS | TABLES |
| DOMAINS | VIEW_COLUMN_USAGE |
| KEY_COLUMN_USAGE | VIEW_TABLE_USAGE |
| PARAMETERS | VIEWS |

Also, some views contain references to different classes of data such as character data or binary data.

When you reference the information schema views, you must use a qualified name that includes the `INFORMATION_SCHEMA` schema name. For example:

```
SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME, COLUMN_DEFAULT
FROM AdventureWorks2012.INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = N'Product';
GO
```

## See Also

System Views (Transact-SQL)
Data Types (Transact-SQL)
System Stored Procedures (Transact-SQL)

# System Stored Procedures (Transact-SQL)

5/3/2018 • 4 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2016) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

In SQL Server 2017, many administrative and informational activities can be performed by using system stored procedures. The system stored procedures are grouped into the categories shown in the following table.

## In This Section

| CATEGORY | DESCRIPTION |
|---|---|
| Active Geo-Replication Stored Procedures | Used to manage to manage Active Geo-Replication configurations in Azure SQL Database |
| Catalog Stored Procedures | Used to implement ODBC data dictionary functions and isolate ODBC applications from changes to underlying system tables. |
| Change Data Capture Stored Procedures | Used to enable, disable, or report on change data capture objects. |
| Cursor Stored Procedures | Used to implements cursor variable functionality. |
| Data Collector Stored Procedures | Used to work with the data collector and the following components: collection sets, collection items, and collection types. |
| Database Engine Stored Procedures | Used for general maintenance of the SQL Server Database Engine. |
| Database Mail Stored Procedures (Transact-SQL) | Used to perform e-mail operations from within an instance of SQL Server. |
| Database Maintenance Plan Stored Procedures | Used to set up core maintenance tasks that are required to manage database performance. |
| Distributed Queries Stored Procedures | Used to implement and manage distributed queries. |
| Filestream and FileTable Stored Procedures (Transact-SQL) | Used to configure and manage the FILESTREAM and FileTable features. |
| Firewall Rules Stored Procedures (Azure SQL Database) | Used to configure the Azure SQL Database firewall. |
| Full-Text Search Stored Procedures | Used to implement and query full-text indexes. |
| General Extended Stored Procedures | Used to provide an interface from an instance of SQL Server to external programs for various maintenance activities. |
| Log Shipping Stored Procedures | Used to configure, modify, and monitor log shipping configurations. |

| CATEGORY | DESCRIPTION |
| --- | --- |
| Management Data Warehouse Stored Procedures (Transact-SQL) | Used to configure the management data warehouse. |
| OLE Automation Stored Procedures | Used to enable standard Automation objects for use within a standard Transact-SQL batch. |
| Policy-Based Management Stored Procedures | Used for Policy-Based Management. |
| PolyBase stored procedures | Add or remove a computer from a PolyBase scale-out group. |
| Query Store Stored Procedures (Transact-SQL) | Used to tune performance. |
| Replication Stored Procedures | Used to manage replication. |
| Security Stored Procedures | Used to manage security. |
| Snapshot Backup Stored Procedures | Used to delete the FILE_SNAPSHOT backup along with all of its snapshots or to delete an individual backup file snapshot. |
| Spatial Index Stored Procedures | Used to analyze and improve the indexing performance of spatial indexes. |
| SQL Server Agent Stored Procedures | Used by SQL Server Profiler to monitor performance and activity. |
| SQL Server Profiler Stored Procedures | Used by SQL Server Agent to manage scheduled and event-driven activities. |
| Stretch Database Stored Procedures | Used to manage stretch databases. |
| Temporal Tables Stored Procedures | Use for temporal tables |
| XML Stored Procedures | Used for XML text management. |

**NOTE**

Unless specifically documented otherwise, all system stored procedures return a value of 0 to indicate success. To indicate failure, a nonzero value is returned.

## API System Stored Procedures

Users that run SQL Server Profiler against ADO, OLE DB, and ODBC applications may notice these applications using system stored procedures that are not covered in the Transact-SQL Reference. These stored procedures are used by the Microsoft SQL Server Native Client OLE DB Provider and the SQL Server Native Client ODBC driver to implement the functionality of a database API. These stored procedures are just the mechanism the provider or driver uses to communicate user requests to an instance of SQL Server. They are intended only for the internal use of the provider or the driver. Calling them explicitly from a SQL Server-based application is not supported.

The sp_createorphan and sp_droporphans stored procedures are used for ODBC **ntext**, **text**, and **image** processing.

The sp_reset_connection stored procedure is used by SQL Server to support remote stored procedure calls in a

transaction. This stored procedure also causes Audit Login and Audit Logout events to fire when a connection is reused from a connection pool.

The system stored procedures in the following tables are used only within an instance of SQL Server or through client APIs and are not intended for general customer use. They are subject to change and compatibility is not guaranteed.

The following stored procedures are documented in SQL Server Books Online:

| | |
|---|---|
| sp_catalogs | sp_column_privileges |
| sp_column_privileges_ex | sp_columns |
| sp_columns_ex | sp_databases |
| sp_cursor | sp_cursorclose |
| sp_cursorexecute | sp_cursorfetch |
| sp_cursoroption | sp_cursoropen |
| sp_cursorprepare | sp_cursorprepexec |
| sp_cursorunprepare | sp_execute |
| sp_datatype_info | sp_fkeys |
| sp_foreignkeys | sp_indexes |
| sp_pkeys | sp_primarykeys |
| sp_prepare | sp_prepexec |
| sp_prepexecrpc | sp_unprepare |
| sp_server_info | sp_special_columns |
| sp_sproc_columns | sp_statistics |
| sp_table_privileges | sp_table_privileges_ex |
| sp_tables | sp_tables_ex |

The following stored procedures are not documented:

| | |
|---|---|
| sp_assemblies_rowset | sp_assemblies_rowset_rmt |
| sp_assemblies_rowset2 | sp_assembly_dependencies_rowset |
| sp_assembly_dependencies_rowset_rmt | sp_assembly_dependencies_rowset2 |

| | |
|---|---|
| sp_bcp_dbcmptlevel | sp_catalogs_rowset |
| sp_catalogs_rowset;2 | sp_catalogs_rowset;5 |
| sp_catalogs_rowset_rmt | sp_catalogs_rowset2 |
| sp_check_constbytable_rowset | sp_check_constbytable_rowset;2 |
| sp_check_constbytable_rowset2 | sp_check_constraints_rowset |
| sp_check_constraints_rowset;2 | sp_check_constraints_rowset2 |
| sp_column_privileges_rowset | sp_column_privileges_rowset;2 |
| sp_column_privileges_rowset;5 | sp_column_privileges_rowset_rmt |
| sp_column_privileges_rowset2 | sp_columns_90 |
| sp_columns_90_rowset | sp_columns_90_rowset_rmt |
| sp_columns_90_rowset2 | sp_columns_ex_90 |
| sp_columns_rowset | sp_columns_rowset;2 |
| sp_columns_rowset;5 | sp_columns_rowset_rmt |
| sp_columns_rowset2 | sp_constr_col_usage_rowset |
| sp_datatype_info_90 | sp_ddopen;1 |
| sp_ddopen;10 | sp_ddopen;11 |
| sp_ddopen;12 | sp_ddopen;13 |
| sp_ddopen;2 | sp_ddopen;3 |
| sp_ddopen;4 | sp_ddopen;5 |
| sp_ddopen;6 | sp_ddopen;7 |
| sp_ddopen;8 | sp_ddopen;9 |
| sp_foreign_keys_rowset | sp_foreign_keys_rowset;2 |
| sp_foreign_keys_rowset;3 | sp_foreign_keys_rowset;5 |
| sp_foreign_keys_rowset_rmt | sp_foreign_keys_rowset2 |
| sp_foreign_keys_rowset3 | sp_indexes_90_rowset |
| sp_indexes_90_rowset_rmt | sp_indexes_90_rowset2 |

| | |
|---|---|
| sp_indexes_rowset | sp_indexes_rowset;2 |
| sp_indexes_rowset;5 | sp_indexes_rowset_rmt |
| sp_indexes_rowset2 | sp_linkedservers_rowset |
| sp_linkedservers_rowset;2 | sp_linkedservers_rowset2 |
| sp_oledb_database | sp_oledb_defdb |
| sp_oledb_deflang | sp_oledb_language |
| sp_oledb_ro_usrname | sp_primary_keys_rowset |
| sp_primary_keys_rowset;2 | sp_primary_keys_rowset;3 |
| sp_primary_keys_rowset;5 | sp_primary_keys_rowset_rmt |
| sp_primary_keys_rowset2 | sp_procedure_params_90_rowset |
| sp_procedure_params_90_rowset2 | sp_procedure_params_rowset |
| sp_procedure_params_rowset;2 | sp_procedure_params_rowset2 |
| sp_procedures_rowset | sp_procedures_rowset;2 |
| sp_procedures_rowset2 | sp_provider_types_90_rowset |
| sp_provider_types_rowset | sp_schemata_rowset |
| sp_schemata_rowset;3 | sp_special_columns_90 |
| sp_sproc_columns_90 | sp_statistics_rowset |
| sp_statistics_rowset;2 | sp_statistics_rowset2 |
| sp_stored_procedures | sp_table_constraints_rowset |
| sp_table_constraints_rowset;2 | sp_table_constraints_rowset2 |
| sp_table_privileges_rowset | sp_table_privileges_rowset;2 |
| sp_table_privileges_rowset;5 | sp_table_privileges_rowset_rmt |
| sp_table_privileges_rowset2 | sp_table_statistics_rowset |
| sp_table_statistics_rowset;2 | sp_table_statistics2_rowset |
| sp_tablecollations | sp_tablecollations_90 |
| sp_tables_info_90_rowset | sp_tables_info_90_rowset_64 |

| | |
|---|---|
| sp_tables_info_90_rowset2 | sp_tables_info_90_rowset2_64 |
| sp_tables_info_rowset | sp_tables_info_rowset;2 |
| sp_tables_info_rowset_64 | sp_tables_info_rowset_64;2 |
| sp_tables_info_rowset2 | sp_tables_info_rowset2_64 |
| sp_tables_rowset;2 | sp_tables_rowset;5 |
| sp_tables_rowset_rmt | sp_tables_rowset2 |
| sp_usertypes_rowset | sp_usertypes_rowset_rmt |
| sp_usertypes_rowset2 | sp_views_rowset |
| sp_views_rowset2 | sp_xml_schema_rowset |
| sp_xml_schema_rowset2 | |

## See Also

CREATE PROCEDURE (Transact-SQL)
Stored Procedures (Database Engine)
Running Stored Procedures (OLE DB)
Running Stored Procedures
Database Engine Stored Procedures (Transact-SQL)
Running Stored Procedures

# System Tables (Transact-SQL)

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2012) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

The topics in this section describe the system tables in SQL Server.

The system tables should not be changed directly by any user. For example, do not try to modify system tables with DELETE, UPDATE, or INSERT statements, or user-defined triggers.

Referencing documented columns in system tables is permissible. However, many of the columns in system tables are not documented. Applications should not be written to directly query undocumented columns. Instead, to retrieve information stored in the system tables, applications should use any one of the following components:

- System stored procedures
- Transact-SQL statements and functions
- SQL Server Management Objects (SMO)
- Replication Management Objects (RMO)
- Database API catalog functions

  These components make up a published API for obtaining system information from SQL Server. Microsoft maintains the compatibility of these components from release to release. The format of the system tables depends upon the internal architecture of SQL Server and may change from release to release. Therefore, applications that directly access the undocumented columns of system tables may have to be changed before they can access a later version of SQL Server.

## In This Section

The system table topics are organized by the following feature areas:

| | |
|---|---|
| Backup and Restore Tables (Transact-SQL) | Log Shipping Tables (Transact-SQL) |
| Change Data Capture Tables (Transact-SQL) | Replication Tables (Transact-SQL) |
| Database Maintenance Plan Tables (Transact-SQL) | SQL Server Agent Tables (Transact-SQL) |
| SQL Server Extended Events Tables (Transact-SQL) | sys.sysoledbusers (Transact-SQL) |
| Integration Services Tables (Transact-SQL) | systranschemas (Transact-SQL) |

## See Also

Compatibility Views (Transact-SQL)
Catalog Views (Transact-SQL)

# Replication Views (Transact-SQL)

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2012) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

These views contain information that is used by replication in Microsoft SQL Server. The views enable easier access to data in replication system tables. Views are created in a user database when that database is enabled as a publication or subscription database. All replication objects are removed from user databases when the database is removed from a replication topology. The preferred method for accessing replication metadata is by using Replication Stored Procedures.

> **IMPORTANT**
> System views should not be altered directly by any user.

## Replication Views

The following is a list of the system views used by replication, grouped by database.

**Replication Views in the msdb Database**

| | |
|---|---|
| MSdatatype_mappings (Transact-SQL) | sysdatatypemappings (Transact-SQL) |

**Replication Views in the Distribution Database**

| | |
|---|---|
| IHextendedArticleView (Transact-SQL) | sysarticles (System View) (Transact-SQL) |
| IHextendedSubscriptionView (Transact-SQL) | sysextendedarticlesview (Transact-SQL) |
| IHsyscolumns (Transact-SQL) | syspublications (System View) (Transact-SQL) |
| MSdistribution_status (Transact-SQL) | syssubscriptions (System View) (Transact-SQL) |
| sysarticlecolumns (System View) (Transact-SQL) | |

**Replication Views in the Publication Database**

| | |
|---|---|
| sysmergeextendedarticlesview (Transact-SQL) | sysmergepartitioninfoview (Transact-SQL) |
| systranschemas (Transact-SQL) | |

**Replication Views in the Subscription Database**

| | |
|---|---|
| sysmergeextendedarticlesview (Transact-SQL) | sysmergepartitioninfoview (Transact-SQL) |

| | |
|---|---|
| | |
| systranschemas (Transact-SQL) | |

## See Also

Replication Tables (Transact-SQL)

# Tables

5/3/2018 • 5 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2016) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Tables are database objects that contain all the data in a database. In tables, data is logically organized in a row-and-column format similar to a spreadsheet. Each row represents a unique record, and each column represents a field in the record. For example, a table that contains employee data for a company might contain a row for each employee and columns representing employee information such as employee number, name, address, job title, and home telephone number.

- The number of tables in a database is limited only by the number of objects allowed in a database (2,147,483,647). A standard user-defined table can have up to 1,024 columns. The number of rows in the table is limited only by the storage capacity of the server.
- You can assign properties to the table and to each column in the table to control the data that is allowed and other properties. For example, you can create constraints on a column to disallow null values or provide a default value if a value is not specified, or you can assign a key constraint on the table that enforces uniqueness or defines a relationship between tables.
- The data in the table can be compressed either by row or by page. Data compression can allow more rows to be stored on a page. For more information, see Data Compression.

## Types of Tables

Besides the standard role of basic user-defined tables, SQL Server provides the following types of tables that serve special purposes in a database.

Partitioned Tables

Partitioned tables are tables whose data is horizontally divided into units which may be spread across more than one filegroup in a database. 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. By default, SQL Server 2017 supports up to 15,000 partitions. For more information, see Partitioned Tables and Indexes.

Temporary Tables

Temporary tables are stored in **tempdb**. There are two types of temporary tables: local and global. They differ from each other in their names, their visibility, and their availability. Local temporary tables have a single number sign (#) as the first character of their names; they are visible only to the current connection for the user, and they are deleted when the user disconnects from the instance of SQL Server. Global temporary tables have two number signs (##) as the first characters of their names; they are visible to any user after they are created, and they are deleted when all users referencing the table disconnect from the instance of SQL Server.

System Tables

SQL Server stores the data that defines the configuration of the server and all its tables in a special set of tables known as system tables. Users cannot directly query or update the system tables. The information in the system tables is made available through the system views. For more information, see System Views (Transact-SQL).

Wide Tables

Wide tables use sparse columns to increase the total of columns that a table can have to 30,000. Sparse columns are ordinary columns that have an optimized storage for null values. Sparse columns reduce the space requirements for null values at the cost of more overhead to retrieve nonnull values. A wide table has defined a

column set, which is an untyped XML representation that combines all the sparse columns of a table into a structured output. The number of indexes and statistics is also increased to 1,000 and 30,000, respectively. The maximum size of a wide table row is 8,019 bytes. Therefore, most of the data in any particular row should be NULL. The maximum number of nonsparse columns plus computed columns in a wide table remains 1,024.

Wide tables have the following performance implications.

- Wide tables can increase the cost to maintain indexes on the table. We recommend that the number of indexes on a wide table be limited to the indexes that are required by the business logic. As the number of indexes increases, so does the DML compile-time and memory requirement. Nonclustered indexes should be filtered indexes that are applied to data subsets. For more information, see Create Filtered Indexes.
- Applications can dynamically add and remove columns from wide tables. When columns are added or removed, compiled query plans are also invalidated. We recommend that you design an application to match the projected workload so that schema changes are minimized.
- When data is added and removed from a wide table, performance can be affected. Applications must be designed for the projected workload so that changes to the table data is minimized.
- Limit the execution of DML statements on a wide table that update multiple rows of a clustering key. These statements can require significant memory resources to compile and execute.
- Switch partition operations on wide tables can be slow and might require large amounts of memory to process. The performance and memory requirements are proportional to the total number of columns in both the source and target partitions.
- Update cursors that update specific columns in a wide table should list the columns explicitly in the FOR UPDATE clause. This will help optimize performance when you use cursors.

## Common Table Tasks

The following table provides links to common tasks associated with creating or modifying a table.

| TABLE TASKS | TOPIC |
| --- | --- |
| Describes how to create a table. | Create Tables (Database Engine) |
| Describes how to delete a table. | Delete Tables (Database Engine) |
| Describes how to create a new table that contains some or all of the columns in an existing table. | Duplicate Tables |
| Describes how to rename a table. | Rename Tables (Database Engine) |
| Describes how to view the properties of the table. | View the Table Definition |
| Describes how to determine whether other objects such as a view or stored procedure depend on a table. | View the Dependencies of a Table |

The following table provides links to common tasks associated with creating or modifying columns in a table.

| COLUMN TASKS | TOPIC |
| --- | --- |
| Describes how to add columns to an existing table. | Add Columns to a Table (Database Engine) |
| Describes how to delete columns from a table. | Delete Columns from a Table |

| COLUMN TASKS | TOPIC |
|---|---|
| Describes how to change the name of a column. | Rename Columns (Database Engine) |
| Describes how to copy columns from one table to another, copying either just the column definition, or the definition and data. | Copy Columns from One Table to Another (Database Engine) |
| Describes how to modify a column definition, by changing the data type or other property. | Modify Columns (Database Engine) |
| Describes how to change the order in which the columns appear. | Change Column Order in a Table |
| Describes how to create a computed column in a table. | Specify Computed Columns in a Table |
| Describes how to specify a default value for a column. This value is used if another value is not supplied. | Specify Default Values for Columns |

## See Also

Primary and Foreign Key Constraints
Unique Constraints and Check Constraints

# Track Data Changes (SQL Server)

5/3/2018 • 11 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2008) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

SQL Server 2017 provides two features that track changes to data in a database: change data capture and change tracking. These features enable applications to determine the DML changes (insert, update, and delete operations) that were made to user tables in a database. Change data capture and change tracking can be enabled on the same database; no special considerations are required. For the editions of SQL Server that support change data capture and change tracking, see Features Supported by the Editions of SQL Server 2016. Change tracking is supported by SQL Database.

## Benefits of Using Change Data Capture or Change Tracking

The ability to query for data that has changed in a database is an important requirement for some applications to be efficient. Typically, to determine data changes, application developers must implement a custom tracking method in their applications by using a combination of triggers, timestamp columns, and additional tables. Creating these applications usually involves a lot of work to implement, leads to schema updates, and often carries a high performance overhead.

Using change data capture or change tracking in applications to track changes in a database, instead of developing a custom solution, has the following benefits:

- There is reduced development time. Because functionality is available in SQL Server 2017, you do not have to develop a custom solution.
- Schema changes are not required. You do not have to add columns, add triggers, or create side table in which to track deleted rows or to store change tracking information if columns cannot be added to the user tables.
- There is a built-in cleanup mechanism. Cleanup for change tracking is performed automatically in the background. Custom cleanup for data that is stored in a side table is not required.
- Functions are provided to obtain change information.
- There is low overhead to DML operations. Synchronous change tracking will always have some overhead. However, using change tracking can help minimize the overhead. The overhead will frequently be less than that of using alternative solutions, especially solutions that require the use of triggers.
- Change tracking is based on committed transactions. The order of the changes is based on transaction commit time. This allows for reliable results to be obtained when there are long-running and overlapping transactions. Custom solutions that use **timestamp** values must be specifically designed to handle these scenarios.
- Standard tools are available that you can use to configure and manage. SQL Server 2017 provides standard DDL statements, SQL Server Management Studio, catalog views, and security permissions.

## Feature Differences Between Change Data Capture and Change Tracking

The following table lists the feature differences between change data capture and change tracking. The tracking mechanism in change data capture involves an asynchronous capture of changes from the transaction log so that

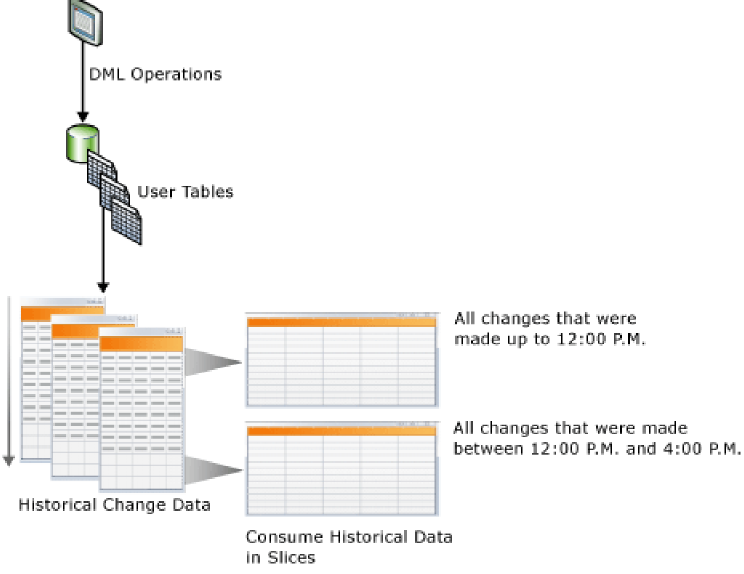
changes are available after the DML operation. In change tracking, the tracking mechanism involves synchronous tracking of changes in line with DML operations so that change information is available immediately.

| FEATURE | CHANGE DATA CAPTURE | CHANGE TRACKING |
| --- | --- | --- |
| **Tracked changes** | | |
| DML changes | Yes | Yes |
| **Tracked information** | | |
| Historical data | Yes | No |
| Whether column was changed | Yes | Yes |
| DML type | Yes | Yes |

# Change Data Capture

Change data capture provides historical change information for a user table by capturing both the fact that DML changes were made and the actual data that was changed. Changes are captured by using an asynchronous process that reads the transaction log and has a low impact on the system.

As shown in the following illustration, the changes that were made to user tables are captured in corresponding change tables. These change tables provide an historical view of the changes over time. The change data capture functions that SQL Server provides enable the change data to be consumed easily and systematically.




### Security Model

This section describes the change data capture security model.

#### Configuration and Administration

To either enable or disable change data capture for a database, the caller of sys.sp_cdc_enable_db (Transact-SQL) or sys.sp_cdc_disable_db (Transact-SQL) must be a member of the fixed server **sysadmin** role. Enabling and disabling change data capture at the table level requires the caller of sys.sp_cdc_enable_table (Transact-SQL) and sys.sp_cdc_disable_table (Transact-SQL) to either be a member of the sysadmin role or a member of the database **database db_owner** role.

Use of the stored procedures to support the administration of change data capture jobs is restricted to members of the server **sysadmin** role and members of the **database db_owner** role.

### Change Enumeration and Metadata Queries

To gain access to the change data that is associated with a capture instance, the user must be granted select access to all the captured columns of the associated source table. In addition, if a gating role is specified when the capture instance is created, the caller must also be a member of the specified gating role. Other general change data capture functions for accessing metadata will be accessible to all database users through the public role, although access to the returned metadata will also typically be gated by using select access to the underlying source tables, and by membership in any defined gating roles.

### DDL Operations to Change Data Capture Enabled Source Tables

When a table is enabled for change data capture, DDL operations can only be applied to the table by a member of the fixed server role **sysadmin**, a member of the **database role db_owner**, or a member of the **database role db_ddladmin**. Users who have explicit grants to perform DDL operations on the table will receive error 22914 if they try these operations.

### Data Type Considerations for Change Data Capture

All base column types are supported by change data capture. The following table lists the behavior and limitations for several column types.

| TYPE OF COLUMN | CHANGES CAPTURED IN CHANGE TABLES | LIMITATIONS |
|---|---|---|
| Sparse Columns | Yes | Does not support capturing changes when using a columnset. |
| Computed Columns | No | Changes to computed columns are not tracked. The column will appear in the change table with the appropriate type, but will have a value of NULL. |
| XML | Yes | Changes to individual XML elements are not tracked. |
| Timestamp | Yes | The data type in the change table is converted to binary. |
| BLOB data types | Yes | The previous image of the BLOB column is stored only if the column itself is changed. |

### Change Data Capture and Other SQL Server Features

This section describes how the following features interact with change data capture:

- Database mirroring
- Transactional replication
- Database restore or attach

#### Database Mirroring

A database that is enabled for change data capture can be mirrored. To ensure that capture and cleanup happen automatically on the mirror, follow these steps:

1. Ensure that SQL Server Agent is running on the mirror.
2. Create the capture job and cleanup job on the mirror after the principal has failed over to the mirror. To create the jobs, use the stored procedure sys.sp_cdc_add_job (Transact-SQL).

   For more information about database mirroring, see Database Mirroring (SQL Server).

#### Transactional Replication

Change data capture and transactional replication can coexist in the same database, but population of the change tables is handled differently when both features are enabled. Change data capture and transactional replication always use the same procedure, sp_replcmds, to read changes from the transaction log. When change data capture is enabled on its own, a SQL Server Agent job calls **sp_replcmds**. When both features are enabled on the same database, the Log Reader Agent calls **sp_replcmds**. This agent populates both the change tables and the distribution database tables. For more information, see Replication Log Reader Agent.

Consider a scenario in which change data capture is enabled on the **AdventureWorks2012** database, and two tables are enabled for capture. To populate the change tables, the capture job calls **sp_replcmds**. The database is enabled for transactional replication, and a publication is created. Now, the Log Reader Agent is created for the database and the capture job is deleted. The Log Reader Agent continues to scan the log from the last log sequence number that was committed to the change table. This ensures data consistency in the change tables. If transactional replication is disabled in this database, the Log Reader Agent is removed and the capture job is re-created.

> **NOTE**
>
> When the Log Reader Agent is used for both change data capture and transactional replication, replicated changes are first written to the distribution database. Then, captured changes are written to the change tables. Both operations are committed together. If there is any latency in writing to the distribution database, there will be a corresponding latency before changes appear in the change tables.

#### Restoring or Attaching a Database Enabled for Change Data Capture

SQL Server uses the following logic to determine if change data capture remains enabled after a database is restored or attached:

- If a database is restored to the same server with the same database name, change data capture remains enabled.
- If a database is restored to another server, by default change data capture is disabled and all related metadata is deleted.

  To retain change data capture, use the **KEEP_CDC** option when restoring the database. For more information about this option, see RESTORE.
- If a database is detached and attached to the same server or another server, change data capture remains enabled.
- If a database is attached or restored with the **KEEP_CDC** option to any edition other than Enterprise, the operation is blocked because change data capture requires SQL Server Enterprise. Error message 932 is displayed:

  ```
  SQL Server cannot load database '%.*ls' because change data capture is enabled. The currently installed
  edition of SQL Server does not support change data capture. Either disable change data capture in the
  database by using a supported edition of SQL Server, or upgrade the instance to one that supports change
  data capture.
  ```

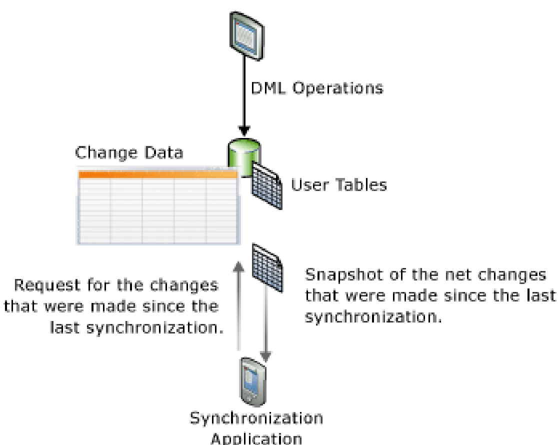
  You can use sys.sp_cdc_disable_db to remove change data capture from a restored or attached database.

## Change Tracking

Change tracking captures the fact that rows in a table were changed, but does not capture the data that was changed. This enables applications to determine the rows that have changed with the latest row data being obtained directly from the user tables. Therefore, change tracking is more limited in the historical questions it can answer compared to change data capture. However, for those applications that do not require the historical information, there is far less storage overhead because of the changed data not being captured. A synchronous tracking mechanism is used to track the changes. This has been designed to have minimal overhead to the DML

operations.

The following illustration shows a synchronization scenario that would benefit by using change tracking. In the scenario, an application requires the following information: all the rows in the table that were changed since the last time that the table was synchronized, and only the current row data. Because a synchronous mechanism is used to track the changes, an application can perform two-way synchronization and reliably detect any conflicts that might have occurred.




**Change Tracking and Sync Services for ADO.NET**

Sync Services for ADO.NET enables synchronization between databases, providing an intuitive and flexible API that enables you to build applications that target offline and collaboration scenarios. Sync Services for ADO.NET provides an API to synchronize changes, but it does not actually track changes in the server or peer database. You can create a custom change tracking system, but this typically introduces significant complexity and performance overhead. To track changes in a server or peer database, we recommend that you use change tracking in SQL Server 2017 because it is easy to configure and provides high performance tracking.

For more information about change tracking and Sync Services for ADO.NET, use the following links:

- About Change Tracking (SQL Server)

  Describes change tracking, provides a high-level overview of how change tracking works, and describes how change tracking interacts with other SQL Server Database Engine features.

- Microsoft Sync Framework Developer Center

  Provides complete documentation for Sync Framework and Sync Services. In the documentation for Sync Services, the topic "How to: Use SQL Server Change Tracking" contains detailed information and code examples.

# Related Tasks (required)

| | |
|---|---|
| **Task** | **Topic** |
| Provides an overview of change data capture. | About Change Data Capture (SQL Server) |
| Describes how to enable and disable change data capture on a database or table. | Enable and Disable Change Data Capture (SQL Server) |
| Describes how to administer and monitor change data capture. | Administer and Monitor Change Data Capture (SQL Server) |

| | |
|---|---|
| Describes how to work with the change data that is available to change data capture consumers. This topic covers validating LSN boundaries, the query functions, and query function scenarios. | Work with Change Data (SQL Server) |
| Provides an overview of change tracking. | About Change Tracking (SQL Server) |
| Describes how to enable and disable change tracking on a database or table. | Enable and Disable Change Tracking (SQL Server) |
| Describes how to manage change tracking, configure security, and determine the effects on storage and performance when change tracking is used. | Manage Change Tracking (SQL Server) |
| Describes how applications that use change tracking can obtain tracked changes, apply these changes to another data store, and update the source database. This topic also describes the role change tracking plays when a failover occurs and a database must be restored from a backup. | Work with Change Tracking (SQL Server) |

## See Also

Change Data Capture Functions (Transact-SQL)
Change Tracking Functions (Transact-SQL)
Change Data Capture Stored Procedures (Transact-SQL)
Change Data Capture Tables (Transact-SQL)
Change Data Capture Related Dynamic Management Views (Transact-SQL)

# Logon Triggers

5/3/2018 • 3 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2008) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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.

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;
```

Note that the LOGON event corresponds to the AUDIT_LOGIN SQL Trace event, which can be used in Event Notifications. The primary difference between triggers and event notifications is that triggers are raised synchronously with events, whereas event notifications are asynchronous. This means, for example, that if you want to stop a session from being established, you must use a logon trigger. An event notification on an AUDIT_LOGIN event cannot be used for this purpose.

## Specifying First and Last Trigger

Multiple triggers can be defined on the LOGON event. Any one of these triggers can be designated the first or last trigger to be fired on an event by using the sp_settriggerorder system stored procedure. SQL Server does not guarantee the execution order of the remaining triggers.

## Managing Transactions

Before SQL Server fires a logon trigger, SQL Server creates an implicit transaction that is independent from any user transaction. Therefore, when the first logon trigger starts firing, the transaction count is 1. After all the logon triggers finish executing, the transaction commits. As with other types of triggers, SQL Server returns an error if a logon trigger finishes execution with a transaction count of 0. The ROLLBACK TRANSACTION statement resets

the transaction count to 0, even if the statement is issued inside a nested transaction. COMMIT TRANSACTION might decrement the transaction count to 0. Therefore, we advise against issuing COMMIT TRANSACTION statements inside logon triggers.

Consider the following when you are using a ROLLBACK TRANSACTION statement inside logon triggers:

- Any data modifications made up to the point of ROLLBACK TRANSACTION are rolled back. These modifications include those made by the current trigger and those made by previous triggers that executed on the same event. Any remaining triggers for the specific event are not executed.
- The current trigger continues to execute any remaining statements that appear after the ROLLBACK statement. If any of these statements modify data, the modifications are not rolled back.

  A user session is not established if any one of the following conditions occur during execution of a trigger on a LOGON event:

- The original implicit transaction is rolled back or fails.
- An error that has severity greater than 20 is raised inside the trigger body.

## Disabling a Logon Trigger

A logon trigger can effectively prevent successful connections to the Database Engine for all users, including members of the **sysadmin** fixed server role. When a logon trigger is preventing connections, members of the **sysadmin** fixed server role can connect by using the dedicated administrator connection, or by starting the Database Engine in minimal configuration mode (-f). For more information, see Database Engine Service Startup Options.

## Related Tasks

| TASK | TOPIC |
| --- | --- |
| Describes how to create logon triggers. Logon triggers can be created from any database, but are registered at the server level and reside in the **master** database. | CREATE TRIGGER (Transact-SQL) |
| Describes how to modify logon triggers. | ALTER TRIGGER (Transact-SQL) |
| Describes how to delete logon triggers. | DROP TRIGGER (Transact-SQL) |
| Describes how to return information about logon triggers. | sys.server_triggers (Transact-SQL)<br>sys.server_trigger_events (Transact-SQL) |
| Describes how to capture logon trigger event data. | |

## See Also

DDL Triggers

# User-Defined Functions

5/3/2018 • 5 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2008) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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**. **Examples.**

**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. **Examples.**

**System Functions**
SQL Server provides many system functions that you can use to perform a variety of operations. They cannot be modified. For more information, see Built-in Functions (Transact-SQL), System Stored Functions (Transact-SQL), and Dynamic Management Views and Functions (Transact-SQL).

## Guidelines

Transact-SQL errors that cause a statement to be canceled and continue with the next statement in the module (such as triggers or stored procedures) are treated differently inside a function. In functions, such errors cause the execution of the function to stop. This in turn causes the statement that invoked the function to be canceled.

The statements in a BEGIN...END block cannot have any side effects. Function side effects are any permanent changes to the state of a resource that has a scope outside the function such as a modification to a database table. The only changes that can be made by the statements in the function are changes to objects local to the function, such as local cursors or variables. Modifications to database tables, operations on cursors that are not local to the function, sending e-mail, attempting a catalog modification, and generating a result set that is returned to the user are examples of actions that cannot be performed in a function.

> **NOTE**
> If a CREATE FUNCTION statement produces side effects against resources that do not exist when the CREATE FUNCTION statement is issued, SQL Server executes the statement. However, SQL Server does not execute the function when it is invoked.

The number of times that a function specified in a query is actually executed can vary between execution plans built by the optimizer. An example is a function invoked by a subquery in a WHERE clause. The number of times the subquery and its function is executed can vary with different access paths chosen by the optimizer.

## Valid statements in a function

The types of statements that are valid in a function include:

- DECLARE statements can be used to define data variables and cursors that are local to the function.
- Assignments of values to objects local to the function, such as using SET to assign values to scalar and table local variables.
- Cursor operations that reference local cursors that are declared, opened, closed, and deallocated in the function. FETCH statements that return data to the client are not allowed. Only FETCH statements that assign values to local variables using the INTO clause are allowed.
- Control-of-flow statements except TRY...CATCH statements.
- SELECT statements containing select lists with expressions that assign values to variables that are local to the function.
- UPDATE, INSERT, and DELETE statements modifying table variables that are local to the function.
- EXECUTE statements calling an extended stored procedure.

**Built-in system functions**

The following nondeterministic built-in functions can be used in Transact-SQL user-defined functions.

| | |
|---|---|
| CURRENT_TIMESTAMP | @@MAX_CONNECTIONS |

| | |
|---|---|
| GET_TRANSMISSION_STATUS | @@PACK_RECEIVED |
| GETDATE | @@PACK_SENT |
| GETUTCDATE | @@PACKET_ERRORS |
| @@CONNECTIONS | @@TIMETICKS |
| @@CPU_BUSY | @@TOTAL_ERRORS |
| @@DBTS | @@TOTAL_READ |
| @@IDLE | @@TOTAL_WRITE |
| @@IO_BUSY | |

The following nondeterministic built-in functions **cannot** be used in Transact-SQL user-defined functions.

| | |
|---|---|
| NEWID | RAND |
| NEWSEQUENTIALID | TEXTPTR |

For a list of deterministic and nondeterministic built-in system functions, see Deterministic and Nondeterministic Functions.

## Schema-bound functions

CREATE FUNCTION supports a SCHEMABINDING clause that binds the function to the schema of any objects it references, such as tables, views, and other user-defined functions. An attempt to alter or drop any object referenced by a schema-bound function fails.

These conditions must be met before you can specify SCHEMABINDING in CREATE FUNCTION:

- All views and user-defined functions referenced by the function must be schema-bound.
- All objects referenced by the function must be in the same database as the function. The objects must be referenced using either one-part or two-part names.
- You must have REFERENCES permission on all objects (tables, views, and user-defined functions) referenced in the function.

  You can use ALTER FUNCTION to remove the schema binding. The ALTER FUNCTION statement should redefine the function without specifying WITH SCHEMABINDING.

## Specifying parameters

A user-defined function takes zero or more input parameters and returns either a scalar value or a table. A function can have a maximum of 1024 input parameters. When a parameter of the function has a default value, the keyword DEFAULT must be specified when calling the function to get the default value. This behavior is different from parameters with default values in user-defined stored procedures in which omitting the parameter also implies the default value. User-defined functions do not support output parameters.

## More examples!

| Task Description | Topic |
|---|---|
| Describes how to create a Transact-SQL user-defined function. | Create User-defined Functions (Database Engine) |
| Describes how create a CLR function. | Create CLR Functions |
| Describes how to create a user-defined aggregate function | Create User-defined Aggregates |
| Describes how to modify a Transact-SQL user-defined function. | Modify User-defined Functions |
| Describes how to delete a user-defined function. | Delete User-defined Functions |
| Describes how to execute a user-defined function. | Execute User-defined Functions |
| Describes how to rename a user-defined function | Rename User-defined Functions |
| Describes how to view the definition of a user-defined function. | View User-defined Functions |

# Views

5/3/2018 • 2 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2008) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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 are generally used to focus, simplify, and customize the perception each user has of the database. 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. For example, you can query the sys.databases catalog view to return information about the user-defined databases available in the instance. For more information, see System Views (Transact-SQL)

## Common View Tasks

The following table provides links to common tasks associated with creating or modifying a view.

| VIEW TASKS | TOPIC |
| --- | --- |

| VIEW TASKS | TOPIC |
|---|---|
| Describes how to create a view. | Create Views |
| Describes how to create an indexed view. | Create Indexed Views |
| Describes how to modify the view definition. | Modify Views |
| Describes how to modify data through a view. | Modify Data Through a View |
| Describes how to delete a view. | Delete Views |
| Describes how to return information about a view such as the view definition. | Get Information About a View |
| Describes how to rename a view. | Rename Views |

## See Also

Create Views over XML Columns
CREATE VIEW (Transact-SQL)

# XML Data (SQL Server)

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server (starting with 2008) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

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.

## In This Section

XML Data Type and Columns (SQL Server)
XML Indexes (SQL Server)
XML Schema Collections (SQL Server)
FOR XML (SQL Server)
OPENXML (Transact-SQL)

## Related Content

Examples of Bulk Import and Export of XML Documents (SQL Server)
XQuery Language Reference (SQL Server)

# Database Engine Tutorials

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

This section contains tutorials for SQL Server 2017 Database Engine.

Tutorial: Getting Started with the Database Engine
This tutorial teaches how to connect to an instance of the Database Engine.

Tutorial: SQL Server Management Studio
This tutorial introduces you to the integrated environment for managing your SQL Server infrastructure. SQL Server Management Studio presents a graphical interface for configuring, monitoring, and administering instances of SQL Server. It also allows you to deploy, monitor, and upgrade the data-tier components used by your applications, such as databases and data warehouses. SQL Server Management Studio also provides Transact-SQL, MDX, DMX, and XML language editors for editing and debugging scripts.

Tutorial: Writing Transact-SQL Statements
This tutorial teaches the fundamental skills of writing the Transact-SQL statements for creating and managing objects in a SQL Server database.

Tutorial: Database Engine Tuning Advisor
This tutorial introduces you to using the advisor to examine how queries are processed, and then review recommendations for improving query performance.

Tutorial: Using the hierarchyid Data Type
This tutorial teaches how to convert a table to a hierarchical structure, and then manage the data in the table.

Tutorial: Signing Stored Procedures with a Certificate
This tutorial illustrates signing stored procedures using a certificate generated by SQL Server.

Tutorial: Ownership Chains and Context Switching
This tutorial uses a scenario to illustrate SQL Server security concepts involving ownership chains and user context switching.

Tutorial: Administering Servers by Using Policy-Based Management
This tutorial teaches how to create policies that enforce site administration standards.

Tutorial: SQL Server Backup and Restore to Windows Azure Blob Storage Service
This tutorial illustrates how to do a SQL Server backup and restore to the Windows Azure Blob Storage Service.

Tutorial: Using the Microsoft Azure Blob storage service with SQL Server 2016 databases
This tutorial helps you understand how to store SQL Server data files in the Windows Azure Blob storage service directly.

## See Also

Tutorials for SQL Server 2016
TechNet WIKI: SQL Server 2012 Samples

# Tutorial: Getting Started with the Database Engine

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Welcome to the Getting Started with the Database Engine tutorial. This tutorial is intended for users who are new to SQL Server and who have installed SQL Server or SQL Server Express. This brief tutorial helps you get started using the Database Engine.

## What You Will Learn

This tutorial shows you how to connect to the Database Engine using SQL Server Management Studio on both the local computer and from another computer.

This tutorial is divided into two lessons:

Lesson 1: Connecting to the Database Engine
In this lesson, you will learn how to connect to the Database Engine and enable additional people to connect.

Lesson 2: Connecting from Another Computer
In this lesson, you will learn how to connect to the Database Engine from a second computer, including enabling protocols, configuring ports, and configuring firewall settings.

## Requirements

This tutorial has no knowledge prerequisites.

Your system must have the following installed to use this tutorial:

- SQL Server Management Studio. To install Management Studio, see Download SQL Server Management Studio.

## See Also

Tutorial: SQL Server Management Studio

# Lesson 1: Connecting to the Database Engine

5/3/2018 • 6 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

> For content related to previous versions of SQL Server, see Lesson 1: Connecting to the Database Engine.

When you install the SQL Server Database Engine, the tools that are installed depend upon the edition and your setup choices. This lesson reviews the principal tools, and shows you how to connect and perform a basic function (authorizing more users).

This lesson contains the following tasks:

- Tools For Getting Started
- Connecting with Management Studio
- Authorizing Additional Connections

## Tools For Getting Started

-The SQL Server Database Engine ships with a variety of tools. This topic describes the first tools you will need, and helps you select the right tool for the job. All tools can be accessed from the **Start** menu. Some tools, such as SQL Server Management Studio, are not installed by default. You must select the tools as part of the client components during setup. For a complete description of the tools described below, search for them in SQL Server Books Online. SQL Server Express contains only a subset of the tools.

#### Basic Tools

- SQL Server Management Studio (SSMS) is the principal tool for administering the Database Engine and writing Transact-SQL code. It is hosted in the Visual Studio shell. SSMS is available as a free download from Microsoft Download Center. The latest version can be used with older versions of the Database Engine.

- SQL Server Configuration Manager installs with both SQL Server and the client tools. It lets you enable server protocols, configure protocol options such as TCP ports, configure server services to start automatically, and configure client computers to connect in your preferred manner. This tool configures the more advanced connectivity elements but does not enable features.

#### Sample Database

The sample databases and samples are not included with SQL Server. Most of the examples that are described in SQL Server Books Online use the **AdventureWorks2012** sample database.

###### To start SQL Server Management Studio

- On current versions of Windows, on the **Start** page, type SSMS, and then click **Microsoft SQL Server Management Studio**.
- When using older versions of Windows, on the **Start** menu, point to **All Programs**, point to **Microsoft SQL Server 2017**, and then click **SQL Server Management Studio**.

###### To start SQL Server Configuration Manager

- On current versions of Windows, on the **Start** page, type **Configuration Manager**, and then click **SQL Server *version* Configuration Manager**.
  -- When using older versions of Windows, on the **Start** menu, point to **All Programs**, point to **Microsoft SQL Server 2017**, point to **Configuration Tools**, and then click **SQL Server Configuration Manager**.
  - ## Connecting with Management Studio

-It is easy to connect to the Database Engine from tools that are running on the same computer if you know the name of the instance, and if you are connecting as a member of the local Administrators group on the computer. The following procedures must be performed on the same computer that hosts SQL Server.

> **NOTE**
> This topic discusses connecting to an on-premises SQL Server. To connect to Azure SQL Database, see Connect to SQL Database with SQL Server Management Studio and execute a sample T-SQL query.

###### To determine the name of the instance of the Database Engine

1. Log into Windows as a member of the Administrators group, and open Management Studio.
2. In the **Connect to Server** dialog box, click **Cancel**.
3. If Registered Servers is not displayed, on the **View** menu, click **Registered Servers**.
4. With **Database Engine** selected on the Registered Servers toolbar, expand **Database Engine**, right-click **Local Server Groups**, point to **Tasks**, and then click **Register Local Servers**. All instances of the Database Engine installed on the computer are displayed. The default instance is unnamed and is shown as the computer name. A named instance displays as the computer name followed by a backward slash (\) and then the name of the instance. For SQL Server Express, the instance is named <*computer_name*>\sqlexpress unless the name was changed during setup.

###### To verify that the Database Engine is running

1. In Registered Servers, if the name of your instance of SQL Server has a green dot with a white arrow next to the name, the Database Engine is running and no further action is necessary.

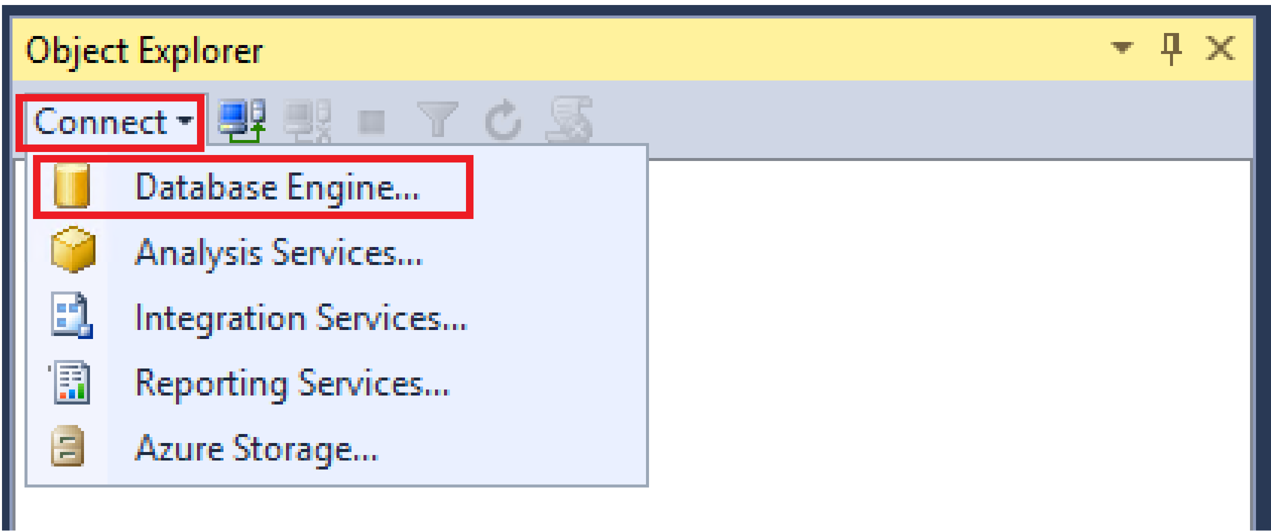
2. If the name of your instance of SQL Server has a red dot with a white square next to the name, the Database Engine is stopped. Right-click the name of the Database Engine, click **Service Control**, and then click **Start**. After a confirmation dialog box, the Database Engine should start and the circle should turn green with a white arrow.

###### To connect to the Database Engine

At least one administrator account was selected when SQL Server was being installed. Perform the following step while logged into Windows as an administrator.

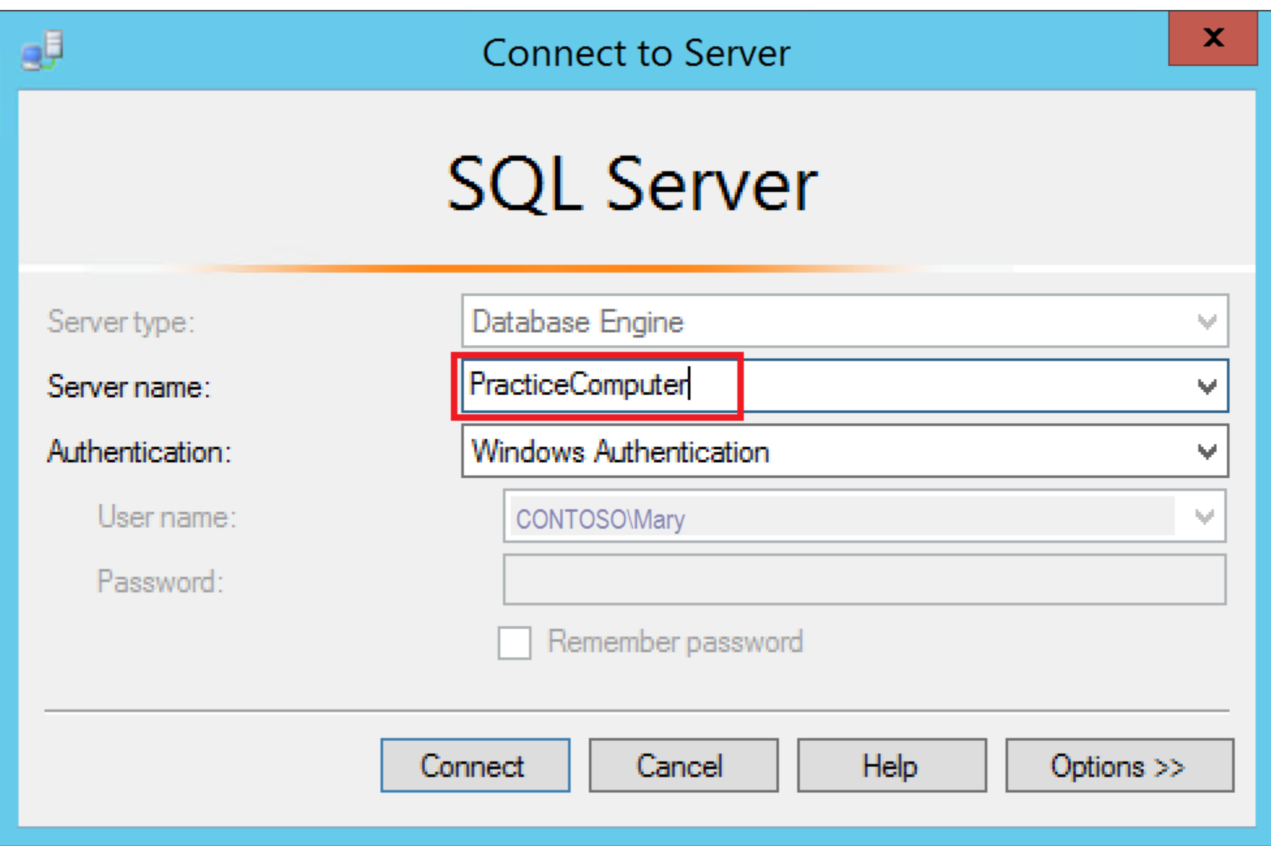
1. In Management Studio, on the **File** menu, click **Connect Object Explorer**.
2. The **Connect to Server** dialog box opens. The **Server type** box displays the type of component that was last used.

3. Select **Database Engine**.




1. In the **Server name** box, type the name of the instance of the Database Engine. For the default instance of SQL

Server, the server name is the computer name. For a named instance of SQL Server, the server name is the *<computer_name>**<instance_name>*, such as **ACCTG_SRVR\SQLEXPRESS**. The following screenshot shows connecting to the default (un-named) instance of SQL Server on a computer named 'PracticeComputer'. The user logged into Windows is Mary from the Contoso domain. When using Windows Authentication you cannot change the user name.




1. Click **Connect**.

> **NOTE**
>
> This tutorial assumes you are new to SQL Server and have no special problems connecting. This should be sufficient for most people and this keeps this tutorial simple. For detailed troubleshooting steps, see Troubleshooting Connecting to the SQL Server Database Engine.

## Authorizing Additional Connections

Now that you have connected to SQL Server as an administrator, one of your first tasks is to authorize other users to connect. You do this by creating a login and authorizing that login to access a database as a user. Logins can be either Windows Authentication logins, which use credentials from Windows, or SQL Server Authentication logins, which store the authentication information in SQL Server and are independent of your Windows credentials. Use Windows Authentication whenever possible.

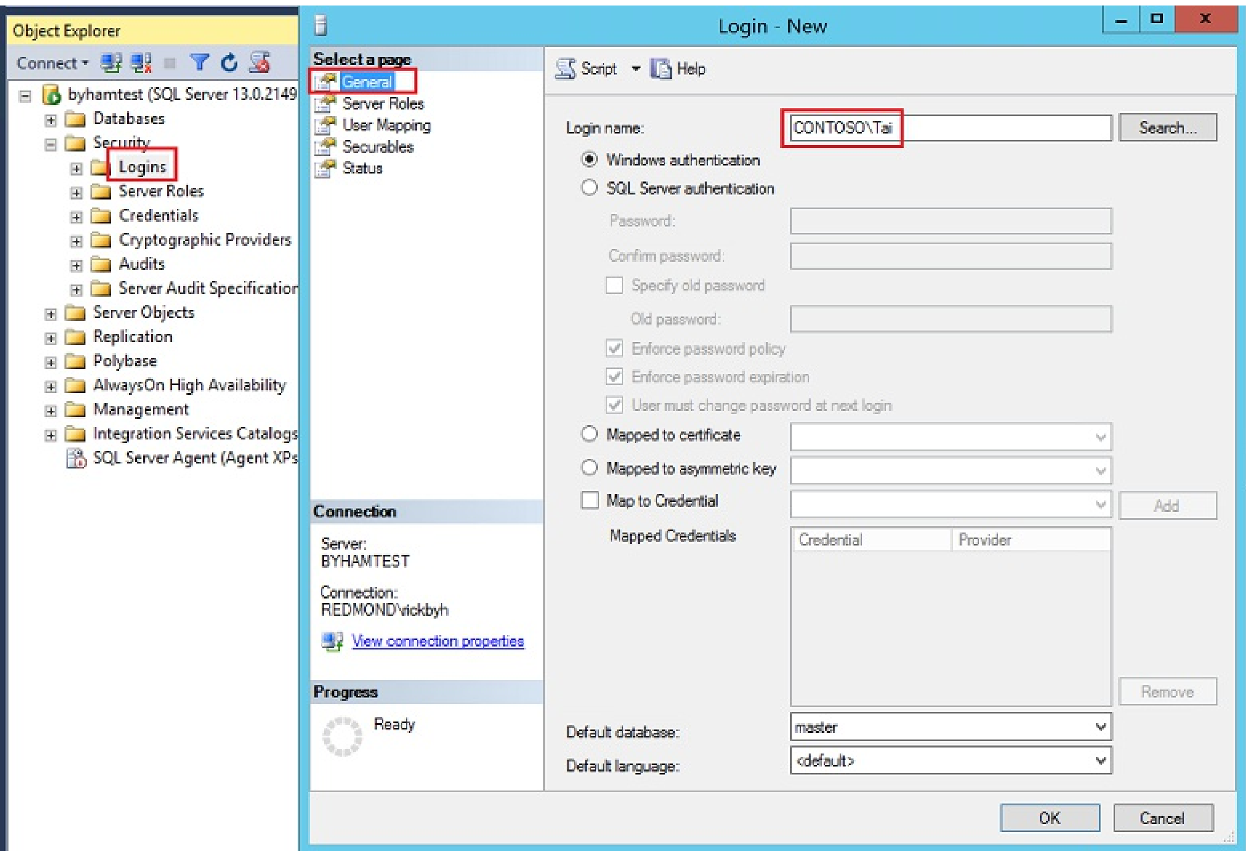
> **TIP**
>
> Most organizations have domain users and will use Windows Authentication. You can experiment on your own, by creating additional local users on your computer. Local users will be authenticated by your computer, so the domain is the computer name. For example if your compute is named `MyComputer` and you create a user named `Test`, then the Windows description of the user is `Mycomputer\Test`.

###### Create a Windows Authentication login

1. In the previous task, you connected to the Database Engine using Management Studio. In Object Explorer,

expand your server instance, expand **Security**, right-click **Logins**, and then click **New Login**. The **Login - New** dialog box appears.

2. On the **General** page, in the **Login name** box, type a Windows login in the format: `<domain>\\<login>`




1. In the **Default database** box, select **AdventureWorks2012** if available. Otherwise select **master**.
2. On the **Server Roles** page, if the new login is to be an administrator, click **sysadmin**, otherwise leave this blank.
3. On the **User Mapping** page, select **Map** for the **AdventureWorks2012** database if it is available. Otherwise select **master**. Note that the **User** box is populated with the login. When closed, the dialog box will create this user in the database.
4. In the **Default Schema** box, type **dbo** to map the login to the database owner schema.
5. Accept the default settings for the **Securables** and **Status** boxes and click **OK** to create the login.

> **IMPORTANT**
>
> This is basic information to get you started. SQL Server provides a rich security environment, and security is obviously an important aspect of database operations.

## Next Lesson

[Lesson 2: Connecting from Another Computer]

# Lesson 2: Connecting from Another Computer

5/3/2018 • 7 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

To enhance security, the Database Engine of SQL Server Developer, Express, and Evaluation editions cannot be accessed from another computer when initially installed. This lesson shows you how to enable the protocols, configure the ports, and configure the Windows Firewall for connecting from other computers.

This lesson contains the following tasks:

- Enabling Protocols
- Configuring a Fixed Port
- Opening Ports in the Firewall
- Connecting to the Database Engine from Another Computer
- Connecting Using the SQL Server Browser Service

## Enabling Protocols

To enhance security, SQL Server Express, Developer, and Evaluation install with only limited network connectivity. Connections to the Database Engine can be made from tools that are running on the same computer, but not from other computers. If you are planning to do your development work on the same computer as the Database Engine, you do not have to enable additional protocols. Management Studio will connect to the Database Engine by using the shared memory protocol. This protocol is already enabled.

If you plan to connect to the Database Engine from another computer, you must enable a protocol, such as TCP/IP.

**How to enable TCP/IP connections from another computer**

1. On the **Start** menu, point to **All Programs**, point to **Microsoft SQL Server 2017**, point to **Configuration Tools**, and then click **SQL Server Configuration Manager**.

   > **NOTE**
   > You might have both 32 bit and 64 bit options available.

   > **NOTE**
   > Because SQL Server Configuration Manager is a snap-in for the Microsoft Management Console program and not a stand-alone program, SQL Server Configuration Manager does not appear as an application in newer versions of Windows. The file name contains a number representing the version number of the SQL Server. To open Configuration Manager from the Run command, here are the paths to the last four versions when Windows is installed on the C drive.

| | |
|---|---|
| SQL Server 2016 | C:\Windows\SysWOW64\SQLServerManager13.msc |
| SQL Server 2014 (12.x) | C:\Windows\SysWOW64\SQLServerManager12.msc |

| | |
|---|---|
| SQL Server 2012 (11.x) | C:\Windows\SysWOW64\SQLServerManager11.msc |
| SQL Server 2008 | C:\Windows\SysWOW64\SQLServerManager10.msc |

2. In **SQL Server Configuration Manager**, expand **SQL Server Network Configuration**, and then click **Protocols for** .

   The default instance (an unnamed instance) is listed as **MSSQLSERVER**. If you installed a named instance, the name you provided is listed. SQL Server 2012 Express installs as **SQLEXPRESS**, unless you changed the name during setup.

3. In the list of protocols, right-click the protocol you want to enable (**TCP/IP**), and then click **Enable**.

> **NOTE**
>
> You must restart the SQL Server service after you make changes to network protocols; however, this is completed in the next task.

## Configuring a Fixed Port

To enhance security, Windows Server 2008, Windows Vista, and Windows 7 all turn on the Windows Firewall. When you want to connect to this instance from another computer, you must open a communication port in the firewall. The default instance of the Database Engine listens on port 1433; therefore, you do not have to configure a fixed port. However, named instances including SQL Server Express listen on dynamic ports. Before you can open a port in the firewall, you must first configure the Database Engine to listen on a specific port known as a fixed port or a static port; otherwise, the Database Engine might listen on a different port each time it is started. For more information about firewalls, the default Windows firewall settings, and a description of the TCP ports that affect the Database Engine, Analysis Services, Reporting Services, and Integration Services, see Configure the Windows Firewall to Allow SQL Server Access.

> **NOTE**
>
> Port number assignments are managed by the Internet Assigned Numbers Authority and are listed at http://www.iana.org. Port numbers should be assigned from numbers 49152 through 65535.

**Configure SQL Server to listen on a specific port**

1. In SQL Server Configuration Manager, expand **SQL Server Network Configuration**, and then click on the server instance you want to configure.

2. In the right pane, double-click **TCP/IP**.

3. In the **TCP/IP Properties** dialog box, click the **IP Addresses** tab.

4. In the **TCP Port** box of the **IPAll** section, type an available port number. For this tutorial, we will use **49172**.

5. Click **OK** to close the dialog box, and click **OK** to the warning that the service must be restarted.

6. In the left pane, click **SQL Server Services**.

7. In the right pane, right-click the instance of SQL Server, and then click **Restart**. When the Database Engine restarts, it will listen on port **49172**.

## Opening Ports in the Firewall

Firewall systems help prevent unauthorized access to computer resources. To connect to SQL Server from another computer when a firewall is on, you must open a port in the firewall.

> **IMPORTANT**
>
> Opening ports in your firewall can leave your server exposed to malicious attacks. Be sure to understand firewall systems before opening ports. For more information, see Security Considerations for a SQL Server Installation.

After you configure the Database Engine to use a fixed port, follow the following instructions to open that port in your Windows Firewall. (You do not have to configure a fixed port for the default instance, because it is already fixed on TCP port 1433.)

**To open a port in the Windows firewall for TCP access (Windows 7)**

1. On the **Start** menu, click **Run**, type **WF.msc**, and then click **OK**.
2. In **Windows Firewall with Advanced Security**, in the left pane, right-click **Inbound Rules**, and then click **New Rule** in the action pane.
3. In the **Rule Type** dialog box, select **Port**, and then click **Next**.
4. In the **Protocol and Ports** dialog box, select **TCP**. Select **Specific local ports**, and then type the port number of the instance of the Database Engine. Type 1433 for the default instance. Type **49172** if you are configuring a named instance and configured a fixed port in the previous task. Click **Next**.
5. In the **Action** dialog box, select **Allow the connection**, and then click **Next**.
6. In the **Profile** dialog box, select any profiles that describe the computer connection environment when you want to connect to the Database Engine, and then click **Next**.
7. In the **Name** dialog box, type a name and description for this rule, and then click **Finish**.

For more information about configuring the firewall including instructions for Windows Vista, see Configure a Windows Firewall for Database Engine Access. For more information about the default Windows firewall settings, and a description of the TCP ports that affect the Database Engine, Analysis Services, Reporting Services, and Integration Services, see Configure the Windows Firewall to Allow SQL Server Access.

## Connecting to the Database Engine from Another Computer

Now that you have configured the Database Engine to listen on a fixed port, and have opened that port in the firewall, you can connect to SQL Server from another computer.

When the SQL Server Browser service is running on the server computer, and when the firewall has opened UDP port 1434, the connection can be made by using the computer name and instance name. To enhance security, our example does not use the SQL Server Browser service.

**To connect to the Database Engine from another computer**

1. On a second computer that contains the SQL Server client tools, log in with an account authorized to connect to SQL Server, and open Management Studio.
2. In the **Connect to Server** dialog box, confirm **Database Engine** in the **Server type** box.
3. In the **Server name** box, type **tcp:** to specify the protocol, followed by the computer name, a comma, and the port number. To connect to the default instance, the port 1433 is implied and can be omitted; therefore, type **tcp:**<*computer_name*>. In our example for a named instance, type **tcp:**<*computer_name*>**,49172**.

> **NOTE**
>
> If you omit **tcp:** from the **Server name** box, then the client will attempt all protocols that are enabled, in the order specified in the client configuration.

4. In the **Authentication** box, confirm **Windows Authentication**, and then click **Connect**.

## Connecting Using the SQL Server Browser Service

The SQL Server Browser service listens for incoming requests for SQL Server resources and provides information about SQL Server instances installed on the computer. When the SQL Server Browser service is running, users can connect to named instances by providing the computer name and instance name, instead of the computer name and port number. Because SQL Server Browser receives unauthenticated UDP requests, it is not always turned on during setup. For a description of the service and an explanation of when it is turned on, see SQL Server Browser Service (Database Engine and SSAS).

To use the SQL Server Browser, you must follow the same steps as before and open UDP port 1434 in the firewall.

This concludes this brief tutorial on basic connectivity.

## Return to Tutorials Portal

Tutorial: Getting Started with the Database Engine

# Tutorial: SQL Server Backup and Restore to Azure Blob Storage Service

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

This tutorial helps you understand how to write backups to and restore from the Azure Blob storage service.

## What you will learn

This tutorial shows you how to create a Storage account, a blob container, creating credentials to access the storage account, writing a backup to the blob service, and performing a simple restore. This tutorial is divided into four lessons:

Lesson 1: Create Azure Storage Objects
In this lesson, you create an Azure storage account and a blob container.

Lesson 2: Create a SQL Server Credential
In this lesson, you create a Credential to store security information used to access the Azure storage account.

Lesson 3: Write a Full Database Backup to the Azure Blob Storage Service
In this lesson, you issue a T-SQL statement to write a backup of the AdventureWorks2012 database to the Azure Blob storage service.

Lesson 4: Perform a Restore From a Full Database Backup
In this lesson, you issue a T-SQL statement to restore from the database backup you created in the previous lesson.

### Requirements

To complete this tutorial, you must be familiar with SQL Server backup and restore concepts and T-SQL syntax. To use this tutorial, your system must meet the following requirements:

- An instance of SQL Server 2017, and AdventureWorks2012 database installed.

  The SQL Server instance can be on-premises or in an Azure Virtual Machine.

  You can use a user database in place of AdventureWorks2012, and modify the tsql syntax accordingly.

- The user account used to issue BACKUP or RESTORE commands should be in the **db_backup operator** database role with **Alter any credential** permissions.

### Additional reading

Following are some recommended reading to understand the concepts, best practices when using Azure Blob storage service for SQL Server backups.

1. SQL Server Backup and Restore with Microsoft Azure Blob Storage Service
2. SQL Server Backup to URL Best Practices and Troubleshooting

## See also

SQL Server Backup and Restore with Microsoft Azure Blob Storage Service

# Tutorial: Signing Stored Procedures with a Certificate

5/3/2018 • 6 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

This tutorial illustrates signing stored procedures using a certificate generated by SQL Server.

> **NOTE**
> To run the code in this tutorial you must have both Mixed Mode security configured and the **AdventureWorks2012** database installed. Scenario

Signing stored procedures using a certificate is useful when you want to require permissions on the stored procedure but you do not want to explicitly grant a user those rights. Although you can accomplish this task in other ways, such as using the EXECUTE AS statement, using a certificate allows you to use a trace to find the original caller of the stored procedure. This provides a high level of auditing, especially during security or Data Definition Language (DDL) operations.

You can create a certificate in the master database to allow server-level permissions, or you can create a certificate in any user databases to allow database-level permissions. In this scenario, a user with no rights to base tables must access a stored procedure in the **AdventureWorks2012** database, and you want to audit the object access trail. Rather than using other ownership chain methods, you will create a server and database user account with no rights to the base objects, and a database user account with rights to a table and a stored procedure. Both the stored procedure and the second database user account will be secured with a certificate. The second database account will have access to all objects, and grant access to the stored procedure to the first database user account.

In this scenario you will first create a database certificate, a stored procedure, and a user, and then you will test the process following these steps:

1. Configure the environment.
2. Create a certificate.
3. Create and sign a stored procedure using the certificate.
4. Create a certificate account using the certificate.
5. Grant the certificate account database rights.
6. Display the access context.
7. Reset the environment.

Each code block in this example is explained in line. To copy the complete example, see Complete Example at the end of this tutorial.

## 1. Configure the Environment

To set the initial context of the example, in SQL Server Management Studio open a new Query and run the following code to open the **AdventureWorks2012** database. This code changes the database context to `AdventureWorks2012` and creates a new server login and database user account ( `TestCreditRatingUser` ), using a password.

```
USE AdventureWorks2012;
GO
-- Set up a login for the test user
CREATE LOGIN TestCreditRatingUser
   WITH PASSWORD = 'ASDECd2439587y'
GO
CREATE USER TestCreditRatingUser
FOR LOGIN TestCreditRatingUser;
GO
```

For more information on the CREATE USER statement, see CREATE USER (Transact-SQL). For more information on the CREATE LOGIN statement, see CREATE LOGIN (Transact-SQL).

## 2. Create a Certificate

You can create certificates in the server using the master database as the context, using a user database, or both. There are multiple options for securing the certificate. For more information on certificates, see CREATE CERTIFICATE (Transact-SQL).

Run this code to create a database certificate and secure it using a password.

```
CREATE CERTIFICATE TestCreditRatingCer
   ENCRYPTION BY PASSWORD = 'pGFD4bb925DGvbd2439587y'
      WITH SUBJECT = 'Credit Rating Records Access',
      EXPIRY_DATE = '12/05/2014';
GO
```

## 3. Create and Sign a Stored Procedure Using the Certificate

Use the following code to create a stored procedure that selects data from the `Vendor` table in the `Purchasing` database schema, restricting access to only the companies with a credit rating of 1. Note that the first section of the stored procedure displays the context of the user account running the stored procedure, which is to demonstrate the concepts only. It is not required to satisfy the requirements.

```
CREATE PROCEDURE TestCreditRatingSP
AS
BEGIN
   -- Show who is running the stored procedure
   SELECT SYSTEM_USER 'system Login'
   , USER AS 'Database Login'
   , NAME AS 'Context'
   , TYPE
   , USAGE
   FROM sys.user_token

   -- Now get the data
   SELECT AccountNumber, Name, CreditRating
   FROM Purchasing.Vendor
   WHERE CreditRating = 1
END
GO
```

Run this code to sign the stored procedure with the database certificate, using a password.

```
ADD SIGNATURE TO TestCreditRatingSP
   BY CERTIFICATE TestCreditRatingCer
    WITH PASSWORD = 'pGFD4bb925DGvbd2439587y';
GO
```

For more information on stored procedures, see Stored Procedures (Database Engine).

For more information on signing stored procedures, see ADD SIGNATURE (Transact-SQL).

## 4. Create a Certificate Account Using the Certificate

Run this code to create a database user ( `TestCreditRatingcertificateAccount` ) from the certificate. This account has no server login, and will ultimately control access to the underlying tables.

```
USE AdventureWorks2012;
GO
CREATE USER TestCreditRatingcertificateAccount
   FROM CERTIFICATE TestCreditRatingCer;
GO
```

## 5. Grant the Certificate Account Database Rights

Run this code to grant `TestCreditRatingcertificateAccount` rights to the base table and the stored procedure.

```
GRANT SELECT
   ON Purchasing.Vendor
   TO TestCreditRatingcertificateAccount;
GO

GRANT EXECUTE
   ON TestCreditRatingSP
   TO TestCreditRatingcertificateAccount;
GO
```

For more information on granting permissions to objects, see GRANT (Transact-SQL).

## 6. Display the Access Context

To display the rights associated with the stored procedure access, run the following code to grant the rights to run the stored procedure to the `TestCreditRatingUser` user.

```
GRANT EXECUTE
   ON TestCreditRatingSP
   TO TestCreditRatingUser;
GO
```

Next, run the following code to run the stored procedure as the dbo login you used on the server. Observe the output of the user context information. It will show the dbo account as the context with its own rights and not through a group membership.

```
EXECUTE TestCreditRatingSP;
GO
```

Run the following code to use the `EXECUTE AS` statement to become the `TestCreditRatingUser` account and run the

stored procedure. This time you will see the user context is set to the USER MAPPED TO CERTIFICATE context.

```
EXECUTE AS LOGIN = 'TestCreditRatingUser';
GO
EXECUTE TestCreditRatingSP;
GO
```

This shows you the auditing available because you signed the stored procedure.

> **NOTE**
>
> Use EXECUTE AS to switch contexts within a database.

## 7. Reset the Environment

The following code uses the `REVERT` statement to return the context of the current account to dbo, and resets the environment.

```
REVERT;
GO
DROP PROCEDURE TestCreditRatingSP;
GO
DROP USER TestCreditRatingcertificateAccount;
GO
DROP USER TestCreditRatingUser;
GO
DROP LOGIN TestCreditRatingUser;
GO
DROP CERTIFICATE TestCreditRatingCer;
GO
```

For more information about the REVERT statement, see REVERT (Transact-SQL).

## Complete Example

This section displays the complete example code.

```
/* Step 1 - Open the AdventureWorks2012 database */
USE AdventureWorks2012;
GO
-- Set up a login for the test user
CREATE LOGIN TestCreditRatingUser
   WITH PASSWORD = 'ASDECd2439587y'
GO
CREATE USER TestCreditRatingUser
FOR LOGIN TestCreditRatingUser;
GO

/* Step 2 - Create a certificate in the AdventureWorks2012 database */
CREATE CERTIFICATE TestCreditRatingCer
   ENCRYPTION BY PASSWORD = 'pGFD4bb925DGvbd2439587y'
      WITH SUBJECT = 'Credit Rating Records Access',
      EXPIRY_DATE = '12/05/2014';
GO

/* Step 3 - Create a stored procedure and
sign it using the certificate */
CREATE PROCEDURE TestCreditRatingSP
AS
BEGIN
```

```
    -- Shows who is running the stored procedure
    SELECT SYSTEM_USER 'system Login'
    , USER AS 'Database Login'
    , NAME AS 'Context'
    , TYPE
    , USAGE
    FROM sys.user_token;

    -- Now get the data
    SELECT AccountNumber, Name, CreditRating
    FROM Purchasing.Vendor
    WHERE CreditRating = 1;
END
GO

ADD SIGNATURE TO TestCreditRatingSP
    BY CERTIFICATE TestCreditRatingCer
     WITH PASSWORD = 'pGFD4bb925DGvbd2439587y';
GO

/* Step 4 - Create a database user for the certificate.
This user has the ownership chain associated with it. */
USE AdventureWorks2012;
GO
CREATE USER TestCreditRatingcertificateAccount
    FROM CERTIFICATE TestCreditRatingCer;
GO

/* Step 5 - Grant the user database rights */
GRANT SELECT
    ON Purchasing.Vendor
    TO TestCreditRatingcertificateAccount;
GO

GRANT EXECUTE
    ON TestCreditRatingSP
    TO TestCreditRatingcertificateAccount;
GO

/* Step 6 - Test, using the EXECUTE AS statement */
GRANT EXECUTE
    ON TestCreditRatingSP
    TO TestCreditRatingUser;
GO

-- Run the procedure as the dbo user, notice the output for the type
EXEC TestCreditRatingSP;
GO

EXECUTE AS LOGIN = 'TestCreditRatingUser';
GO
EXEC TestCreditRatingSP;
GO

/* Step 7 - Clean up the example */
REVERT;
GO
DROP PROCEDURE TestCreditRatingSP;
GO
DROP USER TestCreditRatingcertificateAccount;
GO
DROP USER TestCreditRatingUser;
GO
DROP LOGIN TestCreditRatingUser;
GO
DROP CERTIFICATE TestCreditRatingCer;
GO
```

## See Also

Security Center for SQL Server Database Engine and Azure SQL Database

# Tutorial: Ownership Chains and Context Switching

5/3/2018 • 6 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

This tutorial uses a scenario to illustrate SQL Server security concepts involving ownership chains and user context switching.

> **NOTE**
> To run the code in this tutorial you must have both Mixed Mode security configured and the **AdventureWorks2012** database installed. For more information about Mixed Mode security, see Choose an Authentication Mode.

## Scenario

In this scenario, two users need accounts to access purchase order data stored in the **AdventureWorks2012** database. The requirements are as follows:

- The first account (TestManagerUser) must be able to see all details in every purchase order.
- The second account (TestEmployeeUser) must be able to see the purchase order number, order date, shipping date, product ID numbers, and the ordered and received items per purchase order, by purchase order number, for items where partial shipments have been received.
- All other accounts must retain their current permissions.

To fulfill the requirements of this scenario, the example is broken into four parts that demonstrate the concepts of ownership chains and context switching:

1. Configuring the environment.
2. Creating a stored procedure to access data by purchase order.
3. Accessing the data through the stored procedure.
4. Resetting the environment.

Each code block in this example is explained in line. To copy the complete example, see Complete Example at the end of this tutorial.

## 1. Configure the Environment

Use SQL Server Management Studio and the following code to open the `AdventureWorks2012` database, and use the `CURRENT_USER` Transact-SQL statement to check that the dbo user is displayed as the context.

```
USE AdventureWorks2012;
GO
SELECT CURRENT_USER AS 'Current User Name';
GO
```

For more information about the CURRENT_USER statement, see CURRENT_USER (Transact-SQL).

Use this code as the dbo user to create two users on the server and in the **AdventureWorks2012** database.

```
CREATE LOGIN TestManagerUser
    WITH PASSWORD = '340$Uuxwp7Mcxo7Khx';
GO
CREATE USER TestManagerUser
   FOR LOGIN TestManagerUser
   WITH DEFAULT_SCHEMA = Purchasing;
GO

CREATE LOGIN TestEmployeeUser
    WITH PASSWORD = '340$Uuxwp7Mcxo7Khy';
GO
CREATE USER TestEmployeeUser
   FOR LOGIN TestEmployeeUser;
GO
```

For more information about the CREATE USER statement, see CREATE USER (Transact-SQL). For more information about the CREATE LOGIN statement, see CREATE LOGIN (Transact-SQL).

Use the following code to change the ownership of the `Purchasing` schema to the `TestManagerUser` account. This allows that account to use all Data Manipulation Language (DML) statement access (such as `SELECT` and `INSERT` permissions) on the objects it contains. `TestManagerUser` is also granted the ability to create stored procedures.

```
/* Change owner of the Purchasing Schema to TestManagerUser */
ALTER AUTHORIZATION
   ON SCHEMA::Purchasing
   TO TestManagerUser;
GO

GRANT CREATE PROCEDURE
   TO TestManagerUser
   WITH GRANT OPTION;
GO
```

For more information about the GRANT statement, see GRANT (Transact-SQL). For more information about stored procedures, see Stored Procedures (Database Engine). For a poster of all Database Engine permissions, see https://aka.ms/sql-permissions-poster.

## 2. Create a Stored Procedure to Access Data

To switch context within a database, use the EXECUTE AS statement. EXECUTE AS requires IMPERSONATE permissions.

Use the `EXECUTE AS` statement in the following code to change the context to `TestManagerUser` and create a stored procedure showing only the data required by `TestEmployeeUser`. To satisfy the requirements, the stored procedure accepts one variable for the purchase order number and does not display financial information, and the WHERE clause limits the results to partial shipments.

```
EXECUTE AS LOGIN = 'TestManagerUser'
GO
SELECT CURRENT_USER AS 'Current User Name';
GO

/* Note: The user that calls the EXECUTE AS statement must have IMPERSONATE permissions on the target
principal */
CREATE PROCEDURE usp_ShowWaitingItems @ProductID int
AS
BEGIN
   SELECT a.PurchaseOrderID, a.OrderDate, a.ShipDate
      , b.ProductID, b.OrderQty, b.ReceivedQty
   FROM Purchasing.PurchaseOrderHeader a
      INNER JOIN Purchasing.PurchaseOrderDetail b
         ON a.PurchaseOrderID = b.PurchaseOrderID
   WHERE b.OrderQty > b.ReceivedQty
      AND @ProductID = b.ProductID
   ORDER BY b.ProductID ASC
END
GO
```

Currently `TestEmployeeUser` does not have access to any database objects. The following code (still in the `TestManagerUser` context) grants the user account the ability to query base-table information through the stored procedure.

```
GRANT EXECUTE
   ON OBJECT::Purchasing.usp_ShowWaitingItems
   TO TestEmployeeUser;
GO
```

The stored procedure is part of the `Purchasing` schema, even though no schema was explicitly specified, because `TestManagerUser` is assigned by default to the `Purchasing` schema. You can use system catalog information to locate objects, as shown in the following code.

```
SELECT a.name AS 'Schema'
   , b.name AS 'Object Name'
   , b.type AS 'Object Type'
FROM sys.schemas a
   INNER JOIN sys.objects b
      ON a.schema_id = b.schema_id
WHERE b.name = 'usp_ShowWaitingItems';
GO
```

With this section of the example completed, the code switches context back to dbo using the `REVERT` statement.

```
REVERT;
GO
```

For more information about the REVERT statement, see REVERT (Transact-SQL).

# 3. Access Data Through the Stored Procedure

`TestEmployeeUser` has no permissions on the **AdventureWorks2012** database objects other than a login and the rights assigned to the public database role. The following code returns an error when `TestEmployeeUser` attempts to access base tables.

```
EXECUTE AS LOGIN = 'TestEmployeeUser'
GO
SELECT CURRENT_USER AS 'Current User Name';
GO
/* This won't work */
SELECT *
FROM Purchasing.PurchaseOrderHeader;
GO
SELECT *
FROM Purchasing.PurchaseOrderDetail;
GO
```

Because the objects referenced by the stored procedure created in the last section are owned by `TestManagerUser` by virtue of the `Purchasing` schema ownership, `TestEmployeeUser` can access the base tables through the stored procedure. The following code, still using the `TestEmployeeUser` context, passes purchase order 952 as a parameter.

```
EXEC Purchasing.usp_ShowWaitingItems 952
GO
```

## 4. Reset the Environment

The following code uses the `REVERT` command to return the context of the current account to `dbo`, and then resets the environment.

```
REVERT;
GO
ALTER AUTHORIZATION
ON SCHEMA::Purchasing TO dbo;
GO
DROP PROCEDURE Purchasing.usp_ShowWaitingItems;
GO
DROP USER TestEmployeeUser;
GO
DROP USER TestManagerUser;
GO
DROP LOGIN TestEmployeeUser;
GO
DROP LOGIN TestManagerUser;
GO
```

## Complete Example

This section displays the complete example code.

> **NOTE**
>
> This code does not include the two expected errors that demonstrate the inability of `TestEmployeeUser` to select from base tables.

```
/*
Script:       UserContextTutorial.sql
Author:       Microsoft
Last Updated: Books Online
Conditions:   Execute as DBO or sysadmin in the AdventureWorks database
Section 1:    Configure the Environment
*/
USE AdventureWorks2012;
```

```
GO
SELECT CURRENT_USER AS 'Current User Name';
GO
/* Create server and database users */
CREATE LOGIN TestManagerUser
    WITH PASSWORD = '340$Uuxwp7Mcxo7Khx';

GO

CREATE USER TestManagerUser
   FOR LOGIN TestManagerUser
   WITH DEFAULT_SCHEMA = Purchasing;
GO

CREATE LOGIN TestEmployeeUser
    WITH PASSWORD = '340$Uuxwp7Mcxo7Khy';
GO
CREATE USER TestEmployeeUser
   FOR LOGIN TestEmployeeUser;
GO

/* Change owner of the Purchasing Schema to TestManagerUser */
ALTER AUTHORIZATION
   ON SCHEMA::Purchasing
   TO TestManagerUser;
GO

GRANT CREATE PROCEDURE
   TO TestManagerUser
   WITH GRANT OPTION;
GO

/*
Section 2: Switch Context and Create Objects
*/
EXECUTE AS LOGIN = 'TestManagerUser';
GO
SELECT CURRENT_USER AS 'Current User Name';
GO

/* Note: The user that calls the EXECUTE AS statement must have IMPERSONATE permissions on the target
principal */
CREATE PROCEDURE usp_ShowWaitingItems @ProductID int
AS
BEGIN
   SELECT a.PurchaseOrderID, a.OrderDate, a.ShipDate
      , b.ProductID, b.OrderQty, b.ReceivedQty
   FROM Purchasing.PurchaseOrderHeader AS a
      INNER JOIN Purchasing.PurchaseOrderDetail AS b
         ON a.PurchaseOrderID = b.PurchaseOrderID
   WHERE b.OrderQty > b.ReceivedQty
      AND @ProductID = b.ProductID
   ORDER BY b.ProductID ASC
END;
GO

/* Give the employee the ability to run the procedure */
GRANT EXECUTE
   ON OBJECT::Purchasing.usp_ShowWaitingItems
   TO TestEmployeeUser;
GO

/* Notice that the stored procedure is located in the Purchasing
schema. This also demonstrates system catalogs */
SELECT a.name AS 'Schema'
   , b.name AS 'Object Name'
   , b.type AS 'Object Type'
FROM sys.schemas AS a
   INNER JOIN sys.objects AS b
```

```
      ON a.schema_id = b.schema_id
WHERE b.name = 'usp_ShowWaitingItems';
GO

/* Go back to being the dbo user */
REVERT;
GO

/*
Section 3: Switch Context and Observe Security
*/
EXECUTE AS LOGIN = 'TestEmployeeUser';
GO
SELECT CURRENT_USER AS 'Current User Name';
GO
EXEC Purchasing.usp_ShowWaitingItems 952;
GO

/*
Section 4: Clean Up Example
*/
REVERT;
GO
ALTER AUTHORIZATION
ON SCHEMA::Purchasing TO dbo;
GO
DROP PROCEDURE Purchasing.usp_ShowWaitingItems;
GO
DROP USER TestEmployeeUser;
GO
DROP USER TestManagerUser;
GO
DROP LOGIN TestEmployeeUser;
GO
DROP LOGIN TestManagerUser;
GO
```

## See Also

Security Center for SQL Server Database Engine and Azure SQL Database

# Tutorial: Use Azure Blob storage service with SQL Server 2016

5/3/2018 • 3 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Welcome to the Working with SQL Server 2016 in Microsoft Azure Blob Storage service tutorial. This tutorial helps you understand how to use the Microsoft Azure Blob storage service for SQL Server data files and SQL Server backups.

SQL Server integration support for the Microsoft Azure Blob storage service began as a SQL Server 2012 Service Pack 1 CU2 enhancement, and has been enhanced further with SQL Server 2014 and SQL Server 2016. For an overview of the functionality and benefits of using this functionality, see SQL Server Data Files in Microsoft Azure. For a live demo, see Demo of Point in Time Restore.

**Download**

>> To download SQL Server 2016 (13.x), go to **Evaluation Center**.

>> Have an Azure account? Then go **Here** to spin up a Virtual Machine with SQL Server 2017 already installed.

## What you will learn

This tutorial shows you how to work with SQL Server Data Files in Microsoft Azure Blob storage service in multiple lessons. Each lesson is focused on a specific task and the lessons should be completed in sequence. First, you will learn how to create a new container in Blob storage with a stored access policy and a shared access signature. Then, you will learn how to create a SQL Server credential to integrate SQL Server with Azure blob storage. Next, you will back up a database to Blob storage and restore it to an Azure virtual machine. You will then use SQL Server 2016 file-snapshot transaction log backup to restore to a point in time and to a new database. Finally, the tutorial will demonstrate the use of meta data system stored procedures and functions to help you understand and work with file-snapshot backups.

This article assumes the following:

- You have an on-premises instance of SQL Server 2016 with a copy of AdventureWorks2014 installed.
- You have an Azure storage account.
- You have at least one Azure virtual machines with SQL Server 2016 installed and provisioned this virtual machine in accordance with Provisioning a SQL Server Virtual Machine on Azure. As an option, a second virtual machine can be used for the scenario in Lesson 8. Restore as new database from log backup).

This tutorial is divided into nine lessons, which you must complete in order:

**Lesson 1: Create a stored access policy and a shared access signature on an Azure container**
In this lesson, you create a policy on a new blob container and also generate a shared access signature for use in creating a SQL Server credential.

**Lesson 2: Create a SQL Server credential using a shared access signature**
In this lesson, you create a credential using a SAS key to store security information used to access the Microsoft Azure storage account.

**Lesson 3: Database backup to URL**

In this lesson, you backup an on-premises database to Microsoft Azure Blob storage.

**Lesson 4: Restore database to virtual machine from URL**

In this lesson, you restore a database to an Azure virtual machine from Windows Azure Blob storage.

**Lesson 5: Backup database using file-snapshot backup**

In this lesson, you backup a database using file-snapshot backup and view-file snapshot metadata.

**Lesson 6: Generate activity and backup log using file-snapshot backup**

In this lesson, you generate activity in the database and perform several log backups using file-snapshot backup, and view file-snapshot metadata.

**Lesson 7: Restore a database to a point in time**

In this lesson, you restore a database to a point in time using two file-snapshot log backups.

**Lesson 8. Restore as new database from log backup**

In this lesson, you restore from a file-snapshot log backup to a new database on a different virtual machine.

**Lesson 9: Manage backup sets and file-snapshot backups**

In this lesson, you delete unneeded backup sets and learn how to delete orphaned file snapshot backups (when necessary).

## See Also

SQL Server Data Files in Microsoft Azure
File-Snapshot Backups for Database Files in Azure
SQL Server Backup to URL

# Lesson 1: Create stored access policy and shared access signature

5/3/2018 • 4 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

In this lesson, you will use an Azure PowerShell script to create a shared access signature on an Azure blob container using a stored access policy.

> **NOTE**
> This script is written using Azure PowerShell 5.0.10586.

A shared access signature is a URI that grants restricted access rights to containers, blobs, queues, or tables. A stored access policy provides an additional level of control over shared access signatures on the server side including revoking, expiring, or extending access. When using this new enhancement, you need to create a policy on a container with at least read, write, and list rights.

You can create a stored access policy and a shared access signature by using Azure PowerShell, the Azure Storage SDK, the Azure REST API, or a 3rd party utility. This tutorial demonstrates how to use an Azure PowerShell script to complete this task. The script uses the Resource Manager deployment model and creates the following new resources

- Resource group
- Storage account
- Azure blob container
- SAS policy

This script starts by declaring a number of variables to specify the names for the above resources and the names of the following required input values:

- A prefix name used in naming other resource objects
- Subscription name
- Data center location

> **NOTE**
> This script is written in such a way to allow you to use either existing ARM or classic deployment model resources.

The script completes by generating the appropriate CREATE CREDENTIAL statement that you will use in Lesson 2: Create a SQL Server credential using a shared access signature. This statement is copied to your clipboard for you and is output to the console for you to see.

To create a policy on container and generate a Shared Access Signature (SAS) key, follow these steps:

1. Open Window PowerShell or Windows PowerShell ISE (see version requirements above).

2. Edit and then execute the following script.

```
<#
This script uses the Azure Resource model and creates a new ARM storage account.
Modify this script to use an existing ARM or classic storage account
using the instructions in comments within this script
#>
# Define global variables for the script
$prefixName = '<a prefix name>'  # used as the prefix for the name for various objects
$subscriptionName='<your subscription name>'   # the name  of subscription name you will use
$locationName = '<a data center location>'  # the data center region you will use
$storageAccountName= $prefixName + 'storage' # the storage account name you will create or use
$containerName= $prefixName + 'container'  # the storage container name to which you will attach the
SAS policy with its SAS token
$policyName = $prefixName + 'policy' # the name of the SAS policy

<#
Using Azure Resource Manager deployment model
Comment out this entire section and use the classic storage account name to use an existing classic
storage account
#>

# Set a variable for the name of the resource group you will create or use
$resourceGroupName=$prefixName + 'rg'

# adds an authenticated Azure account for use in the session
Login-AzureRmAccount

# set the tenant, subscription and environment for use in the rest of
Set-AzureRmContext -SubscriptionName $subscriptionName

# create a new resource group - comment out this line to use an existing resource group
New-AzureRmResourceGroup -Name $resourceGroupName -Location $locationName

# Create a new ARM storage account - comment out this line to use an existing ARM storage account
New-AzureRmStorageAccount -Name $storageAccountName -ResourceGroupName $resourceGroupName -Type
Standard_RAGRS -Location $locationName

# Get the access keys for the ARM storage account
$accountKeys = Get-AzureRmStorageAccountKey -ResourceGroupName $resourceGroupName -Name
$storageAccountName

# Create a new storage account context using an ARM storage account
$storageContext = New-AzureStorageContext -StorageAccountName $storageAccountName -StorageAccountKey
$accountKeys[0].Value

<#
Using the Classic deployment model
Use the following four lines to use an existing classic storage account
#>
#Classic storage account name
#Add-AzureAccount
#Select-AzureSubscription -SubscriptionName $subscriptionName #provide an existing classic storage
account
#$accountKeys = Get-AzureStorageKey -StorageAccountName $storageAccountName
#$storageContext = New-AzureStorageContext -StorageAccountName $storageAccountName -StorageAccountKey
$accountKeys.Primary

# The remainder of this script works with either the ARM or classic sections of code above

# Creates a new container in blob storage
$container = New-AzureStorageContainer -Context $storageContext -Name $containerName

# Sets up a Stored Access Policy and a Shared Access Signature for the new container
$policy = New-AzureStorageContainerStoredAccessPolicy -Container $containerName -Policy $policyName -
Context $storageContext -StartTime $(Get-Date).ToUniversalTime().AddMinutes(-5) -ExpiryTime $(Get-
Date).ToUniversalTime().AddYears(10) -Permission rwld
```

```
# Gets the Shared Access Signature for the policy
$sas = New-AzureStorageContainerSASToken -name $containerName -Policy $policyName -Context
$storageContext
Write-Host 'Shared Access Signature= '$($sas.Substring(1))''

# Outputs the Transact SQL to the clipboard and to the screen to create the credential using the Shared
Access Signature
Write-Host 'Credential T-SQL'
$tSql = "CREATE CREDENTIAL [{0}] WITH IDENTITY='Shared Access Signature', SECRET='{1}'" -f
$cbc.Uri,$sas.Substring(1)
$tSql | clip
Write-Host $tSql
```

3. After the script completes, the CREATE CREDENTIAL statement will be in your clipboard for use in the next lesson.

**Next Lesson:**

Lesson 2: Create a SQL Server credential using a shared access signature

## See Also

Shared Access Signatures, Part 1: Understanding the SAS Model
Create Container
Set Container ACL
Get Container ACL

# Lesson 2: Create a SQL Server credential using a shared access signature

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

In this lesson, you will create a credential to store the security information that will be used by SQL Server to write to and read from the Azure container that you created in Lesson 1: Create a stored access policy and a shared access signature on an Azure container.

A SQL Server credential is an object that is used to store authentication information required to connect to a resource outside of SQL Server. The credential stores the URI path of the storage container and the shared access signature for this container.

> **NOTE**
>
> If you wish to backup a SQL Server 2012 SP1 CU2 or later database or a SQL Server 2014 database to this Azure container, you can use the deprecated syntax documented here to create a SQL Server credential based on your storage account key.

## Create SQL Server credential

To create a SQL Server credential, follow these steps:

1. Connect to SQL Server Management Studio.
2. Open a new query windows and connect to the SQL Server 2016 instance of the database engine in your on-premises environment.
3. In the new query window, paste the CREATE CREDENTIAL statement with the shared access signature from Lesson 1 and execute that script.

   The script will look like the following code.

```
USE master
CREATE CREDENTIAL
[https://<mystorageaccountname>.blob.core.windows.net/<mystorageaccountcontainername>] -- this name
must match the container path, start with https and must not contain a forward slash.
   WITH IDENTITY='SHARED ACCESS SIGNATURE' -- this is a mandatory string and do not change it.
   , SECRET = 'sharedaccesssignature' -- this is the shared access signature key that you obtained in
Lesson 1.
GO
```

4. To see all available credentials, you can run the following statement in a query window connected to your instance:

```
SELECT * from sys.credentials
```

5. Open a new query windows and connect to the SQL Server 2016 instance of the database engine in your Azure virtual machine.

6. In the new query window, paste the CREATE CREDENTIAL statement with the shared access signature from Lesson 1 and execute that script.

7. Repeat steps 5 and 6 for any additional SQL Server 2016 instances that you wish to have access to the Azure container.

**Next Lesson:**

Lesson 3: Database backup to URL

# See Also

Credentials (Database Engine)
CREATE CREDENTIAL (Transact-SQL)
sys.credentials (Transact-SQL)

# Lesson 3: Database backup to URL

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

In this lesson, you will back up the AdventureWorks2014 database in your on-premises SQL Server 2016 instance to the Azure container that you created in Lesson 1: Create a stored access policy and a shared access signature on an Azure container.

> **NOTE**
>
> If you wish to backup a SQL Server 2012 SP1 CU2 or later database or a SQL Server 2014 database to this Azure container, you can use the deprecated syntax documented here to backup to URL using the WITH CREDENTIAL syntax.

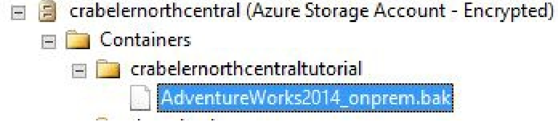
To back up a database to Blob storage, follow these steps:

1. Connect to SQL Server Management Studio.
2. Open a new query window and connect to the SQL Server 2016 instance of the database engine in your Azure virtual machine.
3. Copy and paste the following Transact-SQL script into the query window. Modify the URL appropriately for your storage account name and the container that you specified in Lesson 1 and then execute this script.

```
-- To permit log backups, before the full database backup, modify the database to use the full recovery
model.
USE master;
ALTER DATABASE AdventureWorks2014
   SET RECOVERY FULL;

-- Back up the full AdventureWorks2014 database to the container that you created in Lesson 1
BACKUP DATABASE AdventureWorks2014
   TO URL =
'https://<mystorageaccountname>.blob.core.windows.net/<mystorageaccountcontainername>/AdventureWorks201
4_onprem.bak'
```

4. Open Object Explorer and connect to Azure storage using your storage account and account key.
5. Expand Containers, expand the container that your created in Lesson 1 and verify that the backup from step 3 above appears in this container.




**Next Lesson:**

Lesson 4: Restore database to virtual machine from URL

# Lesson 4: Restore database to virtual machine from URL

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

In this lesson, you will restore the AdventureWorks2014 database to your SQL Server 2016 instance in your Azure virtual machine.

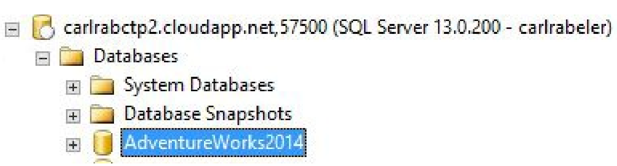
> **NOTE**
> For the purposes of simplicity in this tutorial, we are using the same container for the data and log files that we used for the database backup. In a production environment, you would likely use multiple containers, and frequently multiple data files as well. With SQL Server 2016, you could also consider striping your backup across multiple blobs to increase backup performance when backing up a large database.

To restore the SQL Server 2014 database from Azure blob storage to your SQL Server 2016 instance in your Azure virtual machine, follow these steps:

1. Connect to SQL Server Management Studio.
2. Open a new query window and connect to the SQL Server 2016 instance of the database engine in your Azure virtual machine.
3. Copy and paste the following Transact-SQL script into the query window. Modify the URL appropriately for your storage account name and the container that you specified in Lesson 1 and then execute this script.

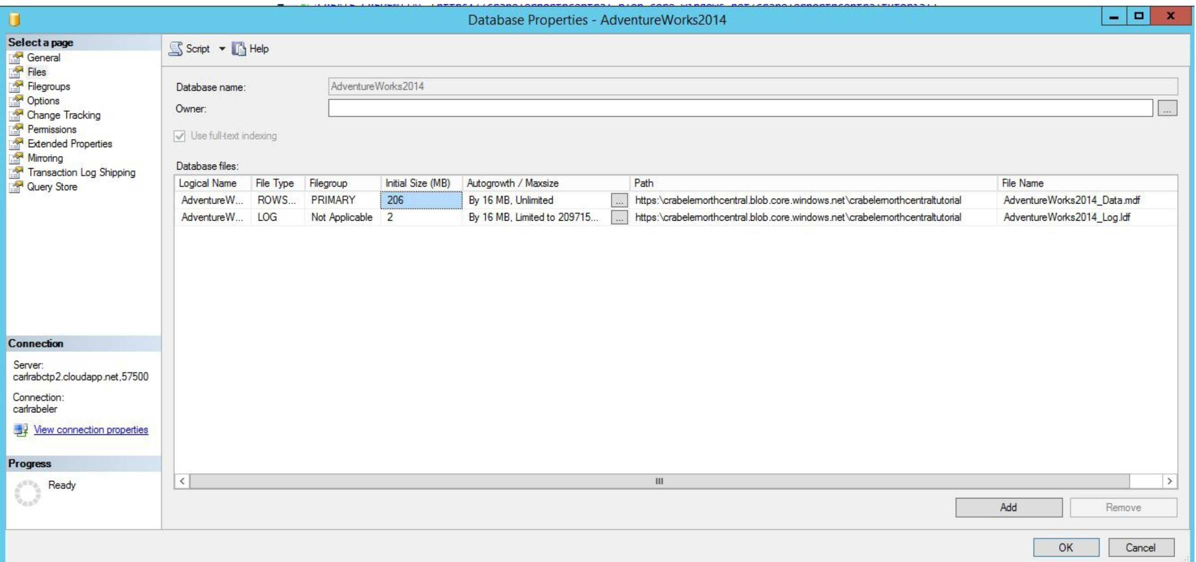
```
-- Restore AdventureWorks2014 from URL to SQL Server instance using Azure blob storage for database files
RESTORE DATABASE AdventureWorks2014
   FROM URL =
'https://<mystorageaccountname>.blob.core.windows.net/<mystorageaccountcontainername>/AdventureWorks2014_onprem.bak'
   WITH
      MOVE 'AdventureWorks2014_data' to
'https://<mystorageaccountname>.blob.core.windows.net/<mystorageaccountcontainername>/AdventureWorks2014_Data.mdf'
     ,MOVE 'AdventureWorks2014_log' to
'https://<mystorageaccountname>.blob.core.windows.net/<mystorageaccountcontainername>/AdventureWorks2014_Log.ldf'
--, REPLACE
```

4. Open Object Explorer and connect to your Azure SQL Server 2016 instance.
5. In Object Explorer, expand the Databases node and verify that the AdventureWorks2014 database has been restored (refresh the node as necessary).



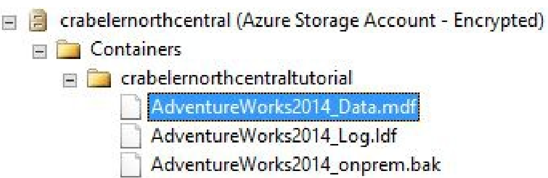
6. In Object Explorer, right-click AdventureWorks2014, and click Properties (click Cancel when done).

7. Click Files and verify that the path for the two database files are URLs pointing to blobs in your Azure blog container.




8. In Object Explorer, connect to Azure storage.

9. Expand Containers, expand the container that your created in Lesson 1 and verify that the AdventureWorks2014_Data.mdf and AdventureWorks2014_Log.ldf from step 3 above appears in this container, along with the backup file from Lesson 3 (refresh the node as necessary).




**Next Lesson:**

Lesson 5: Backup database using file-snapshot backup

# Lesson 5: Backup database using file-snapshot backup

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

In this lesson, you will back up the AdventureWorks2014 database in your Azure virtual machine using file-snapshot backup to perform a nearly instantaneous backup using Azure snapshots. For more information on file-snapshot backups, see File-Snapshot Backups for Database Files in Azure

To back up the AdventureWorks2014 database using file-snapshot backup, follow these steps:

1. Connect to SQL Server Management Studio.

2. Open a new query window and connect to the SQL Server 2016 instance of the database engine in your Azure virtual machine.

3. Copy, paste and execute the following Transact-SQL script into the query window (do not close this query window - you will execute this script again in step 5. This system stored procedure enables you to view the existing file snapshot backups for each file that comprises a specified database. You will notice that there are no file snapshot backups for this database.

```
-- Verify that no file snapshot backups exist
SELECT * FROM sys.fn_db_backup_file_snapshots ('AdventureWorks2014');
```

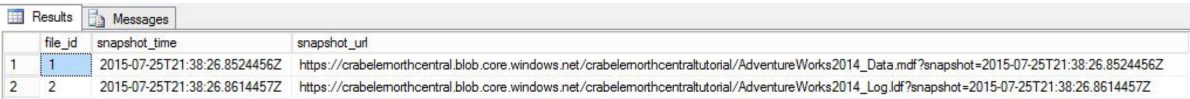
4. Copy and paste the following Transact-SQL script into the query window. Modify the URL appropriately for your storage account name and the container that you specified in Lesson 1 and then execute this script. Notice how quickly this backup occurs.

```
-- Backup the AdventureWorks2014 database with FILE_SNAPSHOT
BACKUP DATABASE AdventureWorks2014
   TO URL =
'https://<mystorageaccountname>.blob.core.windows.net/<mystorageaccountcontainername>/AdventureWorks201
4_Azure.bak'
   WITH FILE_SNAPSHOT;
```

```
Messages
Processed 0 pages for database 'AdventureWorks2014', file 'AdventureWorks2014_Data' on file 1.
Processed 0 pages for database 'AdventureWorks2014', file 'AdventureWorks2014_Log' on file 1.
BACKUP DATABASE successfully processed 0 pages in 0.331 seconds (0.000 MB/sec).
```

5. After verifying that the script in step 4 executed successfully, execute the following script again. Notice that the file-snapshot backup operation in step 4 generated file-snapshots of both the data and log file.

```
-- Verify that two file-snapshot backups exist
SELECT * FROM sys.fn_db_backup_file_snapshots ('AdventureWorks2014');
```

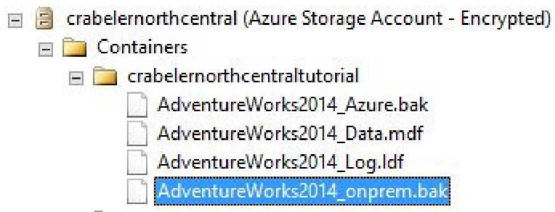


| | file_id | snapshot_time | snapshot_url |
|---|---|---|---|
| 1 | 1 | 2015-07-25T21:38:26.8524456Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-25T21:38:26.8524456Z |
| 2 | 2 | 2015-07-25T21:38:26.8614457Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-25T21:38:26.8614457Z |

6. In Object Explorer, in your SQL Server 2016 instance in your Azure virtual machine, expand the Databases node and verify that the AdventureWorks2014 database has been restored to this instance (refresh the node as necessary).

7. In Object Explorer, connect to Azure storage.

8. Expand Containers, expand the container that you created in Lesson 1 and verify that the AdventureWorks2014_Azure.bak from step 4 above appears in this container, along with the backup file from Lesson 3 and the database files from Lesson 4 (refresh the node as necessary).




**Next Lesson:**

Lesson 6: Generate activity and backup log using file-snapshot backup

# See Also

File-Snapshot Backups for Database Files in Azure
sys.fn_db_backup_file_snapshots (Transact-SQL)

# Lesson 6: Generate activity and backup log using file-snapshot backup

5/3/2018 • 2 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

In this lesson, you will generate activity in the AdventureWorks2014 database and periodically create transaction log backups using file-snapshot backups. For more information on using file snapshot backups, see File-Snapshot Backups for Database Files in Azure.

To generate activity in the AdventureWorks2014 database and periodically create transaction log backups using file-snapshot backups, follow these steps:

1. Connect to SQL Server Management Studio.

2. Open two new query windows and connect each to the SQL Server 2016 instance of the database engine in your Azure virtual machine.

3. Copy, paste and execute the following Transact-SQL script into one of the query windows. Notice that the Production.Location table has 14 rows before we add new rows in step 4.

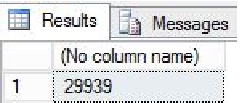
```
-- Verify row count at start
SELECT COUNT (*) from AdventureWorks2014.Production.Location;
```

4. Copy and paste the following two Transact-SQL scripts into the two separate query windows. Modify the URL appropriately for your storage account name and the container that you specified in Lesson 1 and then execute these scripts simultaneously in separate query windows. These scripts will take about seven minutes to complete.

```
-- Insert 30,000 new rows into the Production.Location table in the AdventureWorks2014 database in
batches of 75
DECLARE @count INT=1, @inner INT;
WHILE @count < 400
   BEGIN
      BEGIN TRAN;
         SET @inner =1;
            WHILE @inner <= 75
               BEGIN;
                  INSERT INTO AdventureWorks2014.Production.Location
                     (Name, CostRate, Availability, ModifiedDate)
                        VALUES (NEWID(), .5, 5.2, GETDATE());
                  SET @inner = @inner + 1;
               END;
      COMMIT;
   WAITFOR DELAY '00:00:01';
   SET @count = @count + 1;
   END;
SELECT COUNT (*) from AdventureWorks2014.Production.Location;
```

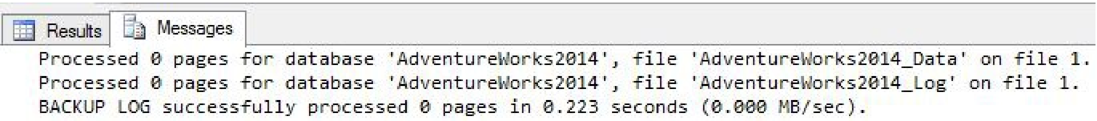
```
--take 7 transaction log backups with FILE_SNAPSHOT, one per minute, and include the row count and the
execution time in the backup file name
DECLARE @count INT=1, @device NVARCHAR(120), @numrows INT;
WHILE @count <= 7
   BEGIN
         SET @numrows = (SELECT COUNT (*) FROM AdventureWorks2014.Production.Location);
         SET @device =
'https://<mystorageaccountname>.blob.core.windows.net/<mystorageaccountcontainername>/tutorial-' +
CONVERT (varchar(10),@numrows) + '-' + FORMAT(GETDATE(), 'yyyyMMddHHmmss') + '.bak';
         BACKUP LOG AdventureWorks2014 TO URL = @device WITH FILE_SNAPSHOT;
         SELECT * from sys.fn_db_backup_file_snapshots ('AdventureWorks2014');
      WAITFOR DELAY '00:1:00';
         SET @count = @count + 1;
   END;
```

5. Review the output of the first script and notice that final row count is now 29,939.

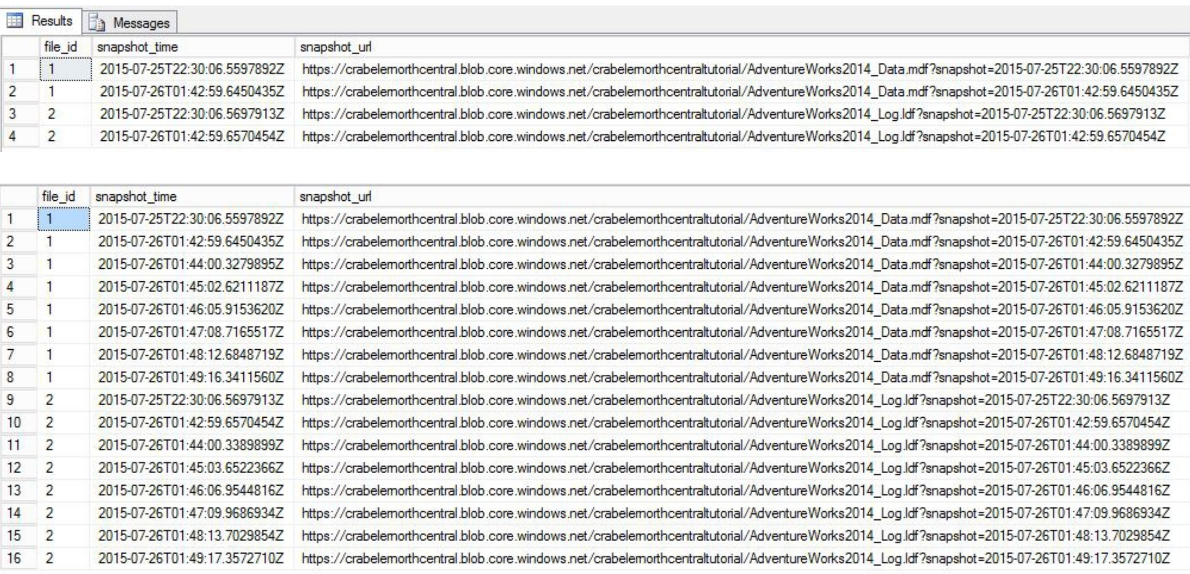


| | (No column name) |
|---|---|
| 1 | 29939 |

6. Review the output of the second script and notice that each time the BACKUP LOG statement is executed that two new file snapshots are created, one file snapshot of the log file and one file snapshot of the data file - for a total of two file snapshots for each database file. After the second script completes, notice that there are now a total of 16 file snapshots, 8 for each database file - one from the BACKUP DATABASE statement and one for each execution of the BACKUP LOG statement.







| | file_id | snapshot_time | snapshot_url |
|---|---|---|---|
| 1 | 1 | 2015-07-25T22:30:06.5597892Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-25T22:30:06.5597892Z |
| 2 | 1 | 2015-07-26T01:42:59.6450435Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-26T01:42:59.6450435Z |
| 3 | 2 | 2015-07-25T22:30:06.5697913Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-25T22:30:06.5697913Z |
| 4 | 2 | 2015-07-26T01:42:59.6570454Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-26T01:42:59.6570454Z |

| | file_id | snapshot_time | snapshot_url |
|---|---|---|---|
| 1 | 1 | 2015-07-25T22:30:06.5597892Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-25T22:30:06.5597892Z |
| 2 | 1 | 2015-07-26T01:42:59.6450435Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-26T01:42:59.6450435Z |
| 3 | 1 | 2015-07-26T01:44:00.3279895Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-26T01:44:00.3279895Z |
| 4 | 1 | 2015-07-26T01:45:02.6211187Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-26T01:45:02.6211187Z |
| 5 | 1 | 2015-07-26T01:46:05.9153620Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-26T01:46:05.9153620Z |
| 6 | 1 | 2015-07-26T01:47:08.7165517Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-26T01:47:08.7165517Z |
| 7 | 1 | 2015-07-26T01:48:12.6848719Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-26T01:48:12.6848719Z |
| 8 | 1 | 2015-07-26T01:49:16.3411560Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-26T01:49:16.3411560Z |
| 9 | 2 | 2015-07-25T22:30:06.5697913Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-25T22:30:06.5697913Z |
| 10 | 2 | 2015-07-26T01:42:59.6570454Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-26T01:42:59.6570454Z |
| 11 | 2 | 2015-07-26T01:44:00.3389899Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-26T01:44:00.3389899Z |
| 12 | 2 | 2015-07-26T01:45:03.6522366Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-26T01:45:03.6522366Z |
| 13 | 2 | 2015-07-26T01:46:06.9544816Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-26T01:46:06.9544816Z |
| 14 | 2 | 2015-07-26T01:47:09.9686934Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-26T01:47:09.9686934Z |
| 15 | 2 | 2015-07-26T01:48:13.7029854Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-26T01:48:13.7029854Z |
| 16 | 2 | 2015-07-26T01:49:17.3572710Z | https://crabelemorthcentral.blob.core.windows.net/crabelemorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-26T01:49:17.3572710Z |

7. In Object Explorer, connect to Azure storage.

8. Expand Containers, expand the container that your created in Lesson 1 and verify that 7 new backup files appear, along with the blobs from the previous lessons (refresh the node as needed).

crabelernorthcentral (Azure Storage Account - Encrypted)
Containers
crabelernorthcentraltutorial
AdventureWorks2014_Azure.bak
AdventureWorks2014_Data.mdf
AdventureWorks2014_Log.ldf
AdventureWorks2014_onprem.bak
tutorial-13814-20150726014602.bak
tutorial-18314-20150726014706.bak
tutorial-22889-20150726014808.bak
tutorial-27464-20150726014912.bak
tutorial-4964-20150726014358.bak
tutorial-614-20150726014257.bak
tutorial-9389-20150726014458.bak

**Next Lesson:**

Lesson 7: Restore a database to a point in time

# Lesson 7: Restore a database to a point in time

5/3/2018 • 2 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server  Azure SQL Database  Azure SQL Data Warehouse  Parallel Data Warehouse

In this lesson, you will restore the AdventureWorks2014 database to a point in time between two of the transaction log backups.

With traditional backups, to accomplish point in time restore, you would need to use the full database backup, perhaps a differential backup, and all of the transaction log files up to and just past the point in time to which you wish to restore. With file-snapshot backups, you only need the two adjacent log backup files that provide the goal posts framing the time to which you wish to restore. You only need two log file snapshot backup sets because each log backup creates a file snapshot of each database file (each data file and the log file).

To restore a database to a specified point in time from file snapshot backup sets, follow these steps:

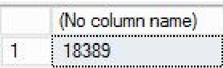
1. Connect to SQL Server Management Studio.
2. Open a new query window and connect to the SQL Server 2016 instance of the database engine in your Azure virtual machine.
3. Copy, paste and execute the following Transact-SQL script into the query window. Verify that the Production.Location table has 29,939 rows before we restore it to a point in time when there are fewer rows in step 5.

```
-- Verify row count at start
SELECT COUNT (*) from AdventureWorks2014.Production.Location
```

4. Copy and paste the following Transact-SQL script into the query window. Select two adjacent log backup files and convert the file name into the date and time you will need for this script. Modify the URL appropriately for your storage account name and the container that you specified in Lesson 1, provide the first and second backup file names, provide the STOPAT time in the format of "June 26, 2015 01:48 PM" and then execute this script. This script will take a few minutes to complete

```
-- restore and recover to a point in time between the times of two transaction log backups, and then
verify the row count
ALTER DATABASE AdventureWorks2014 SET SINGLE_USER WITH ROLLBACK IMMEDIATE;
RESTORE DATABASE AdventureWorks2014
   FROM URL =
'https://<mystorageaccountname>.blob.core.windows.net/<mystorageaccountcontainername>/<firstbackupfile>
.bak'
   WITH NORECOVERY,REPLACE;
RESTORE LOG AdventureWorks2014
   FROM URL =
'https://<mystorageaccountname>.blob.core.windows.net/<mystorageaccountcontainername>/<secondbackupfile
>.bak'
   WITH RECOVERY, STOPAT = 'June 26, 2015 01:48 PM';
ALTER DATABASE AdventureWorks2014 set multi_user;
-- get new count
SELECT COUNT (*) FROM AdventureWorks2014.Production.Location ;
```

5. Review the output. Notice that after the restore the row count is 18,389, which is a row count number between log backup 5 and 6 (your row count will vary).



| | (No column name) |
|---|---|
| 1 | 18389 |

**Next Lesson:**

Lesson 8. Restore as new database from log backup

# Lesson 8. Restore as new database from log backup

5/3/2018 • 2 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

In this lesson, you will restore the AdventureWorks2014 database as a new database from a file-snapshot transaction log backup.

In this scenario, you are performing a restore to a SQL Server instance on a different virtual machine for the purposes of business analysis and reporting. Restoring to a different instance on a different virtual machine offloads the workload to a virtual machine dedicated and sized for this purpose, removing its resource requirements from the transactional system.

Restore from a transaction log backup with file-snapshot backup is very quick, substantially quicker than with traditional streaming backups. With traditional streaming backups, you would need to use the full database backup, perhaps a differential backup, and some or all of the transaction log backups (or a new full database backup). However, with file-snapshot log backups, you only need the most recent log backup (or any other log backup or any two adjacent log backups for point in time restore to a point between two log backup times). To be clear, you only need one log file-snapshot backup set because each file-snapshot log backup creates a file snapshot of each database file (each data file and the log file).

To restore a database to a new database from a transaction log backup using file snapshot backup, follow these steps:

1. Connect to SQL Server Management Studio.

2. Open a new query window and connect to the SQL Server 2016 instance of the database engine in an Azure virtual machine.

   > **NOTE**
   >
   > If this is a different Azure virtual machine than you have been using for the previous lessons, make sure you have followed the steps in Lesson 2: Create a SQL Server credential using a shared access signature. If you wish to restore to a different container, follow the steps in Lesson 1: Create a stored access policy and a shared access signature on an Azure container and Lesson 2: Create a SQL Server credential using a shared access signature for the new container.

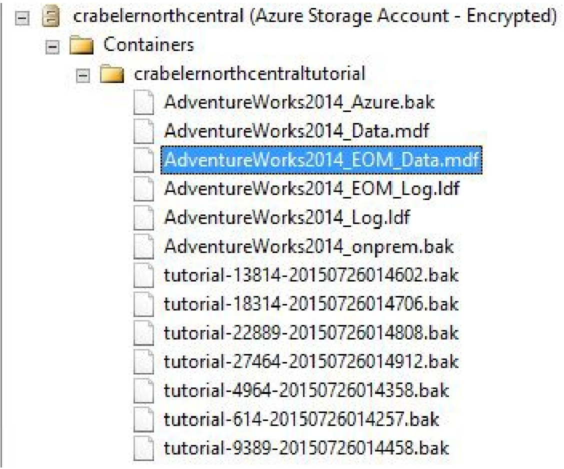
3. Copy and paste the following Transact-SQL script into the query window. Select the log backup file you wish to use. Modify the URL appropriately for your storage account name and the container that you specified in Lesson 1, provide the log backup file name and then execute this script.

```
-- restore as a new database from a transaction log backup file
RESTORE DATABASE AdventureWorks2014_EOM
    FROM URL =
'https://<mystorageaccountname>.blob.core.windows.net/<mystorageaccountcontainername>/<logbackupfile.ba
k'
    WITH MOVE 'AdventureWorks2014_data' to
'https://<mystorageaccountname>.blob.core.windows.net/<mystorageaccountcontainername>/AdventureWorks201
4_EOM_Data.mdf'
   , MOVE 'AdventureWorks2014_log' to
'https://<mystorageaccountname>.blob.core.windows.net/<mystorageaccountcontainername>/AdventureWorks201
4_EOM_Log.ldf'
   , RECOVERY
--, REPLACE
```

4. Review the output to verify the restore was successful.

5. In Object Explorer, connect to Azure storage.

6. Expand Containers, expand the container that your created in Lesson 1 (refresh if necessary) and verify that the new data and log files appear in the container, along with the blobs from the previous lessons.




Lesson 9: Manage backup sets and file-snapshot backups

# Lesson 9: Manage backup sets and file-snapshot backups

5/3/2018 • 1 min to read • Edit Online

**THIS TOPIC APPLIES TO:** SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

In this lesson, you will delete a backup set using the sp_delete_backup (Transact-SQL) system stored procedure. This system stored procedure deletes the backup file and the file snapshot on each database file associated with this backup set.

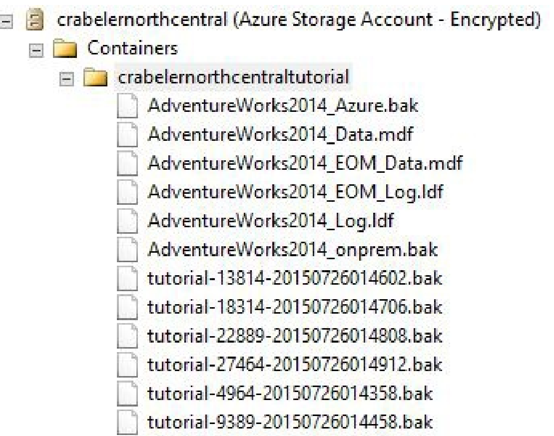
> **NOTE**
>
> If you attempt to delete a backup set by simply deleting the backup file from the Azure blob container, you will only delete the backup file itself - the associated file snapshots will remain. If you find yourself in this scenario, use the sys.fn_db_backup_file_snapshots (Transact-SQL) system function to identify the URL of the orphaned file snapshots and use the sp_delete_backup_file_snapshot (Transact-SQL) system stored procedure to delete each orphaned file snapshot. For more information, see File-Snapshot Backups for Database Files in Azure.

To delete a file-snapshot backup set, follow these steps:

1. Connect to SQL Server Management Studio.
2. Open a new query window and connect to the SQL Server 2016 instance of the database engine in your Azure virtual machine (or to any SQL Server 2016 instance with permissions to read and write on this container).
3. Copy and paste the following Transact-SQL script into the query window. Select the log backup you wish to delete along with its associated file snapshots. Modify the URL appropriately for your storage account name and the container that you specified in Lesson 1, provide the log backup file name and then execute this script.

```
sys.sp_delete_backup
'https://<mystorageaccountname>.blob.core.windows.net/<mystorageaccountcontainername>/tutorial-9164-
20150726012420.bak';
```

4. In Object Explorer, connect to Azure storage.
5. Expand Containers, expand the container that your created in Lesson 1 and verify that the backup file you used in step 3 no longer appears in this container (refresh the node as necessary).




6. Copy, paste and execute the following Transact-SQL script into the query window to verify that two file snapshots have been deleted.

```
-- verify that two file snapshots have been removed
SELECT * from sys.fn_db_backup_file_snapshots ('AdventureWorks2014');
```

Results | Messages

| | file_id | snapshot_time | snapshot_url |
|---|---|---|---|
| 1 | 1 | 2015-07-25T22:30:06.5597892Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-25T22:30:06.5597892Z |
| 2 | 1 | 2015-07-26T01:44:00.3279895Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-26T01:44:00.3279895Z |
| 3 | 1 | 2015-07-26T01:45:02.6211187Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-26T01:45:02.6211187Z |
| 4 | 1 | 2015-07-26T01:46:05.9153620Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-26T01:46:05.9153620Z |
| 5 | 1 | 2015-07-26T01:47:08.7165517Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-26T01:47:08.7165517Z |
| 6 | 1 | 2015-07-26T01:48:12.6848719Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-26T01:48:12.6848719Z |
| 7 | 1 | 2015-07-26T01:49:16.3411560Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Data.mdf?snapshot=2015-07-26T01:49:16.3411560Z |
| 8 | 2 | 2015-07-25T22:30:06.5697913Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-25T22:30:06.5697913Z |
| 9 | 2 | 2015-07-26T01:44:00.3389899Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-26T01:44:00.3389899Z |
| 10 | 2 | 2015-07-26T01:45:03.6522366Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-26T01:45:03.6522366Z |
| 11 | 2 | 2015-07-26T01:46:06.9544816Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-26T01:46:06.9544816Z |
| 12 | 2 | 2015-07-26T01:47:09.9686934Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-26T01:47:09.9686934Z |
| 13 | 2 | 2015-07-26T01:48:13.7029854Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-26T01:48:13.7029854Z |
| 14 | 2 | 2015-07-26T01:49:17.3572710Z | https://crabelernorthcentral.blob.core.windows.net/crabelernorthcentraltutorial/AdventureWorks2014_Log.ldf?snapshot=2015-07-26T01:49:17.3572710Z |

**End of Tutorial**

# See Also

File-Snapshot Backups for Database Files in Azure
sp_delete_backup (Transact-SQL)
sys.fn_db_backup_file_snapshots (Transact-SQL)
sp_delete_backup_file_snapshot (Transact-SQL)