Database Training

Microsoft SQL Server

Day 3

SummitWorks™
GLOBAL SOLUTION ARCHITECTS

Authorized & published by Summitworks Technologies Inc

Agenda

- Indexes
  - Clustered
  - Non-Clustered
  - Create, Alter and Drop
  - Uses
- Transaction Management
  - Introduction
  - Begin
  - Commit
  - Rollback
  - Save

- Error handling and Try Catch
- Cursors and uses
- Triggers

  Purpose of triggers

  Differences between Stored Procedure, User defined Function and Triggers
- Dynamic SQL
- Temp variable and Temp tables
- Query Execution Plan(Planed and Actual)

Database Backup and Restore options, create backup plans

Indexes

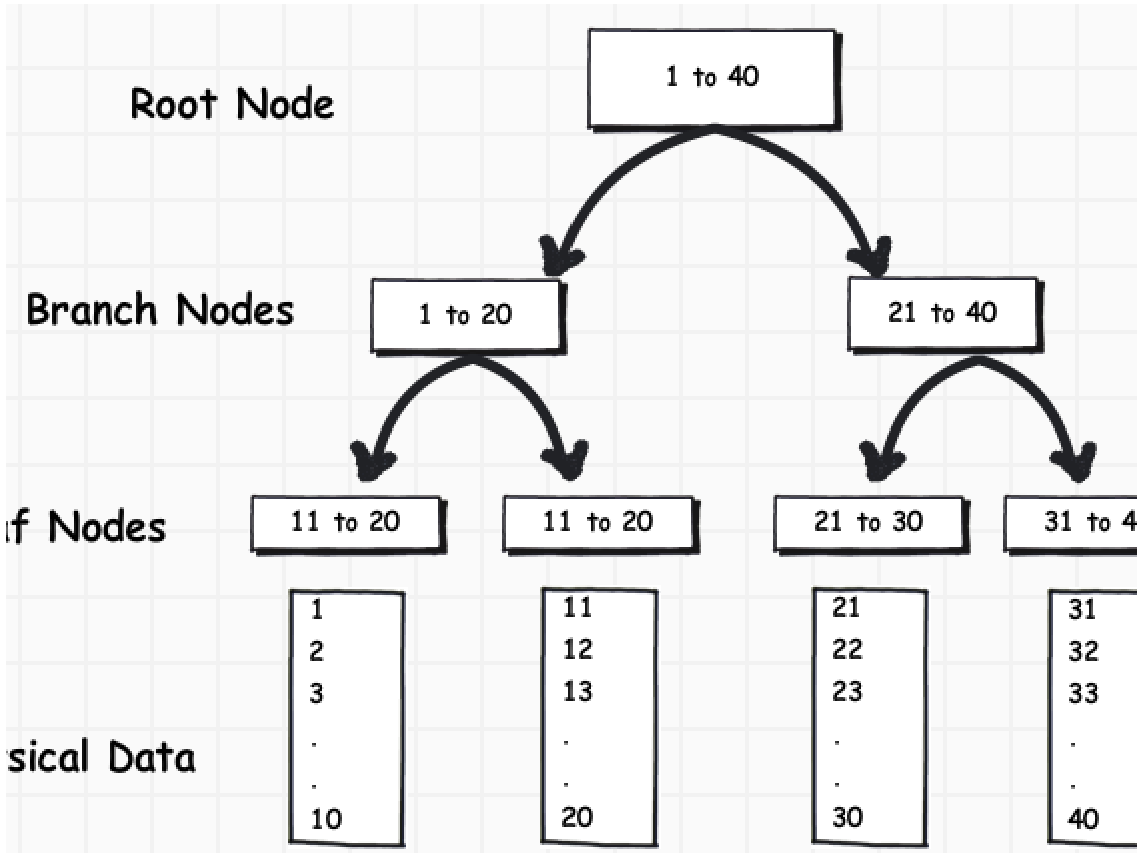
- Indexes are special lookup tables that the database search engine can use to speed up data retrieval.
- For example, if you want to reference all pages in a book that discuss a certain topic, you first refer to the index, which lists all topics alphabetically and are then referred to one or more specific page numbers.
- Indexes are created on columns in tables or views .
- Indexes cannot be created on columns with datatypes as **image**, **text,** and **varchar(max)**.
- Creating an index involves the CREATE INDEX statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in ascending or descending order.
- An index can be dropped using SQL DROP command. Care should be taken when dropping an index because performance may be slowed or improved.

Indexes

- An index is made up of a set of pages (index nodes) that are organized in a B-tree structure.
- This structure is hierarchical in nature, with the root node at the top of the hierarchy and the leaf nodes at the bottom.

- There are two types of Indexes :
    - Clustered
    - Non-Clustered

The B - Tree



Unique Index

- A unique index is one in which no two rows are permitted to have the same index key value. A clustered index on a view must be unique.

- The Database Engine does not allow creating a unique index on columns that already include duplicate values, whether or not IGNORE_DUP_KEY is set to ON. If this is tried, the Database Engine displays an error message.

- Duplicate values must be removed before a unique index can be created on the column or columns. Columns that are used in a unique index should be set to NOT NULL, because multiple null values are considered duplicates when a unique index is created.

Clustered Index

- A Clustered index physically stores rows of data on the leaf node.
- The data can be either sorted in ascending or descending order.
- There can only be one clustered index per table.
- A table with a clustered index is called a clustered table. A table with no clustered index is a ‘heap’.

Non- Clustered Index

- Here the leaf nodes of a non-clustered index contain the values from the indexed columns and pointers to data rows, rather than contain the data rows themselves.
- This required the query to take an additional step in order to locate the actual data.
- Non-clustered indexes cannot be sorted like clustered indexes. It is possible to have more than one non-clustered index per table.

Create and Alter an Index

- CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index_name

  ON <object> (column [ASC | DESC] [,...n])

  [INCLUDE (column_name [,...n])]

  [WHERE <filter_predicate>]

  [WITH (<relational_index_option> [,...n])] [;]

```
CREATE UNIQUE| CLUSTERED | NONCLUSTERED INDEX index_name
on table_name (column_name);
```

- ALTER INDEX { index_name | ALL }

  ON <object> { REBUILD

  [WITH (<rebuild_index_option> [,...n])] | DISABLE | REORGANIZE [PARTITION = partition_number]

  [WITH (<reorganize_option>)] | SET (<set_index_option> [,...n]) } [;]

Drop an Index

- Removes one or more relational, spatial or filtered indexes from the current database.

- The DROP INDEX statement does not apply to indexes created by defining PRIMARY KEY or UNIQUE constraints. To remove the constraint and corresponding index, use ALTER TABLE with the DROP CONSTRAINT clause.

- DROP INDEX [IF EXISTS] { <drop_relational_or_xml_or_spatial_index> [,...n] | <drop_backward_compatible_index> [,...n] }

```
DROP INDEX index_name;
```

Benefits of Indexes

Uses of Indexes

- Indexes are used to speed up database searches.
- Indexes are also used to speed up the performance of queries.

When should indexes be avoided?

- Although indexes are intended to enhance a database's performance, there are times when they should be avoided. The following guidelines indicate when the use of an index should be reconsidered:
- Indexes should not be used on small tables.
- Tables that have frequent, large batch update or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

Transaction Management

- Transactions group a set of tasks into a single execution unit. Each transaction begins with a specific task and ends when all the tasks in the group successfully complete. If any of the tasks fails, the transaction fails. Therefore, a transaction has only two results: success or failure. Incomplete steps result in the failure of the transaction.
- Users can group two or more Transact-SQL statements into a single transaction using the following statements:
    - Begin Transaction
    - Rollback Transaction
    - Commit Transaction
- If anything goes wrong with any of the grouped statements, all changes need to be aborted. The process of reversing changes is called **rollback** in SQL Server terminology. If everything is in order with all statements within a single transaction, all changes are recorded together in the database. In SQL Server terminology, we say that these changes are **committed** to the database.

Properties of Transactions

Transactions have the following four standard properties, usually referred to by the acronym ACID:

- **Atomicity:** ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure, and previous operations are rolled back to their former state.
- **Consistency:** ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation:** enables transactions to operate independently of and transparent to each other.
- **Durability:** ensures that the result or effect of a committed transaction persists in case of a system failure.

Begin Transaction

- Marks the starting point of an explicit, local transaction. BEGIN TRANSACTION increments @@TRANCOUNT by 1.

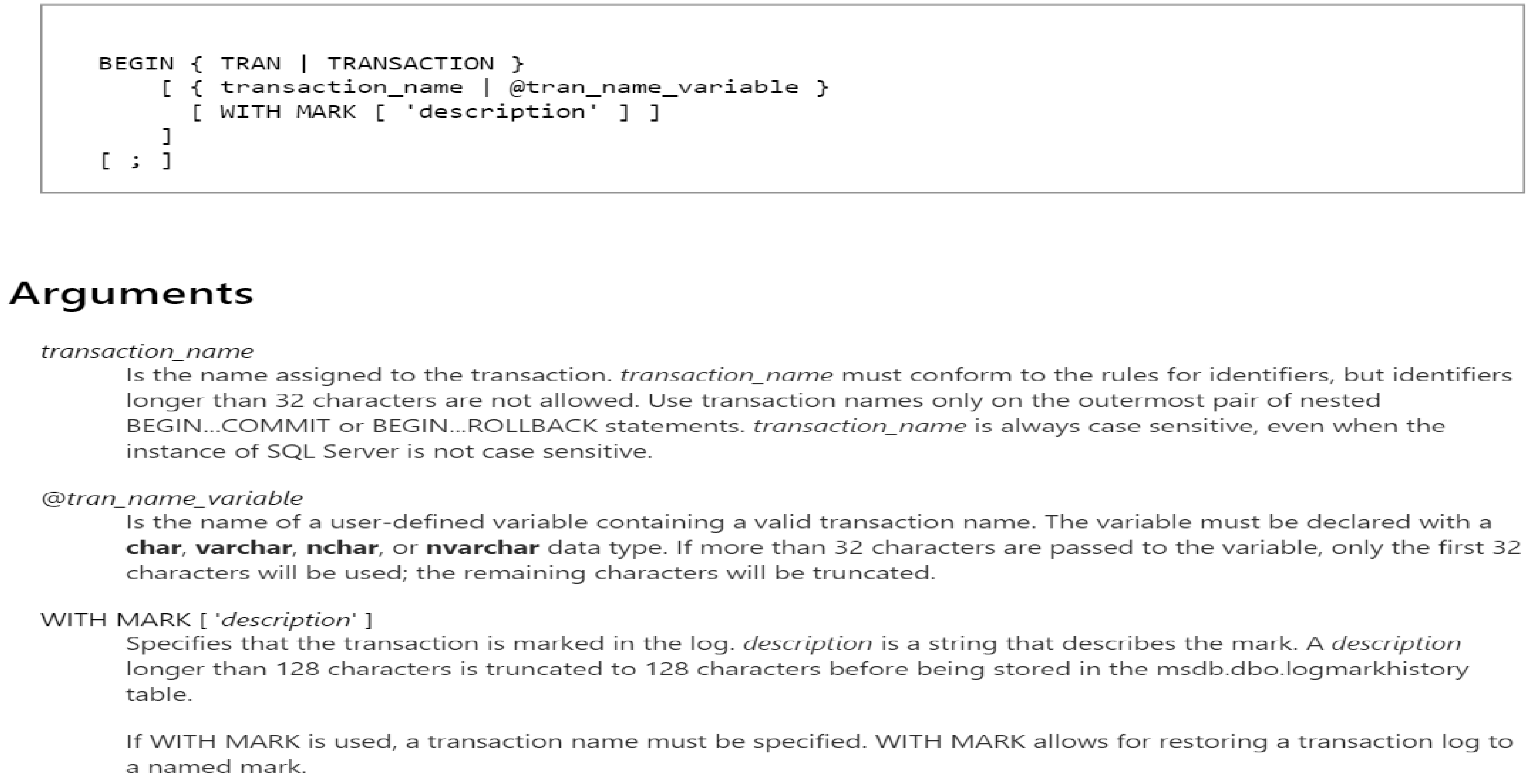
- BEGIN TRANSACTION represents a point at which the data referenced by a connection is logically and physically consistent. If errors are encountered, all data modifications made after the BEGIN TRANSACTION can be rolled back to return the data to this known state of consistency. Each transaction lasts until either it completes without errors and COMMIT TRANSACTION is issued to make the modifications a permanent part of the database, or errors are encountered and all modifications are erased with a ROLLBACK TRANSACTION statement.

Begin Transaction

- BEGIN TRANSACTION starts a local transaction for the connection issuing the statement. Depending on the current transaction isolation level settings, many resources acquired to support the Transact-SQL statements issued by the connection are locked by the transaction until it is completed with either a COMMIT TRANSACTION or ROLLBACK TRANSACTION statement. Transactions left outstanding for long periods of time can prevent other users from accessing these locked resources, and also can prevent log truncation.

- Although BEGIN TRANSACTION starts a local transaction, it is not recorded in the transaction log until the application subsequently performs an action that must be recorded in the log, such as executing an INSERT, UPDATE, or DELETE statement. An application can perform actions such as acquiring locks to protect the transaction isolation level of SELECT statements, but nothing is recorded in the log until the application performs a modification action.

Begin Transaction



```
BEGIN { TRAN | TRANSACTION }
    [ { transaction_name | @tran_name_variable }
      [ WITH MARK [ 'description' ] ]
    ]
[ ; ]
```

Arguments

transaction_name

Is the name assigned to the transaction. *transaction_name* must conform to the rules for identifiers, but identifiers longer than 32 characters are not allowed. Use transaction names only on the outermost pair of nested BEGIN...COMMIT or BEGIN...ROLLBACK statements. *transaction_name* is always case sensitive, even when the instance of SQL Server is not case sensitive.

@tran_name_variable

Is the name of a user-defined variable containing a valid transaction name. The variable must be declared with a **char**, **varchar**, **nchar**, or **nvarchar** data type. If more than 32 characters are passed to the variable, only the first 32 characters will be used; the remaining characters will be truncated.

WITH MARK ['*description*']

Specifies that the transaction is marked in the log. *description* is a string that describes the mark. A *description* longer than 128 characters is truncated to 128 characters before being stored in the msdb.dbo.logmarkhistory table.

If WITH MARK is used, a transaction name must be specified. WITH MARK allows for restoring a transaction log to a named mark.

Rollback Transaction

The ROLLBACK Command:

- The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database.
- The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.
- A ROLLBACK TRANSACTION statement does not produce any messages to the user. If warnings are needed in stored procedures or triggers, use the RAISERROR or PRINT statements. RAISERROR is the preferred statement for indicating errors.

Syntax

```
ROLLBACK { TRAN | TRANSACTION }
     [ transaction_name | @tran_name_variable
     | savepoint_name | @savepoint_variable ]
[ ; ]
```

Arguments

transaction_name
Is the name assigned to the transaction on BEGIN TRANSACTION. *transaction_name* must conform to the rules for identifiers, but only the first 32 characters of the transaction name are used. When nesting transactions, *transaction_name* must be the name from the outermost BEGIN TRANSACTION statement. *transaction_name* is always case sensitive, even when the instance of SQL Server is not case sensitive.

@tran_name_variable
Is the name of a user-defined variable containing a valid transaction name. The variable must be declared with a **char**, **varchar**, **nchar**, or **nvarchar** data type.

savepoint_name
Is *savepoint_name* from a SAVE TRANSACTION statement. *savepoint_name* must conform to the rules for identifiers. Use *savepoint_name* when a conditional rollback should affect only part of the transaction.

@savepoint_variable
Is name of a user-defined variable containing a valid savepoint name. The variable must be declared with a **char**, **varchar**, **nchar**, or **nvarchar** data type.

- ROLLBACK TRANSACTION without a *savepoint_name* or *transaction_name* rolls back to the beginning of the transaction. When nesting transactions, this same statement rolls back all inner transactions to the outermost BEGIN TRANSACTION statement. In both cases, ROLLBACK TRANSACTION decrements the @@TRANCOUNT system function to 0. ROLLBACK TRANSACTION *savepoint_name* does not decrement @@TRANCOUNT.

- ROLLBACK TRANSACTION cannot reference a *savepoint_name* in distributed transactions started either explicitly with BEGIN DISTRIBUTED TRANSACTION or escalated from a local transaction.

- A transaction cannot be rolled back after a COMMIT TRANSACTION statement is executed, except when the COMMIT TRANSACTION is associated with a nested transaction that is contained within the transaction being rolled back. In this instance, the nested transaction will also be rolled back, even if you have issued a COMMIT TRANSACTION for it.

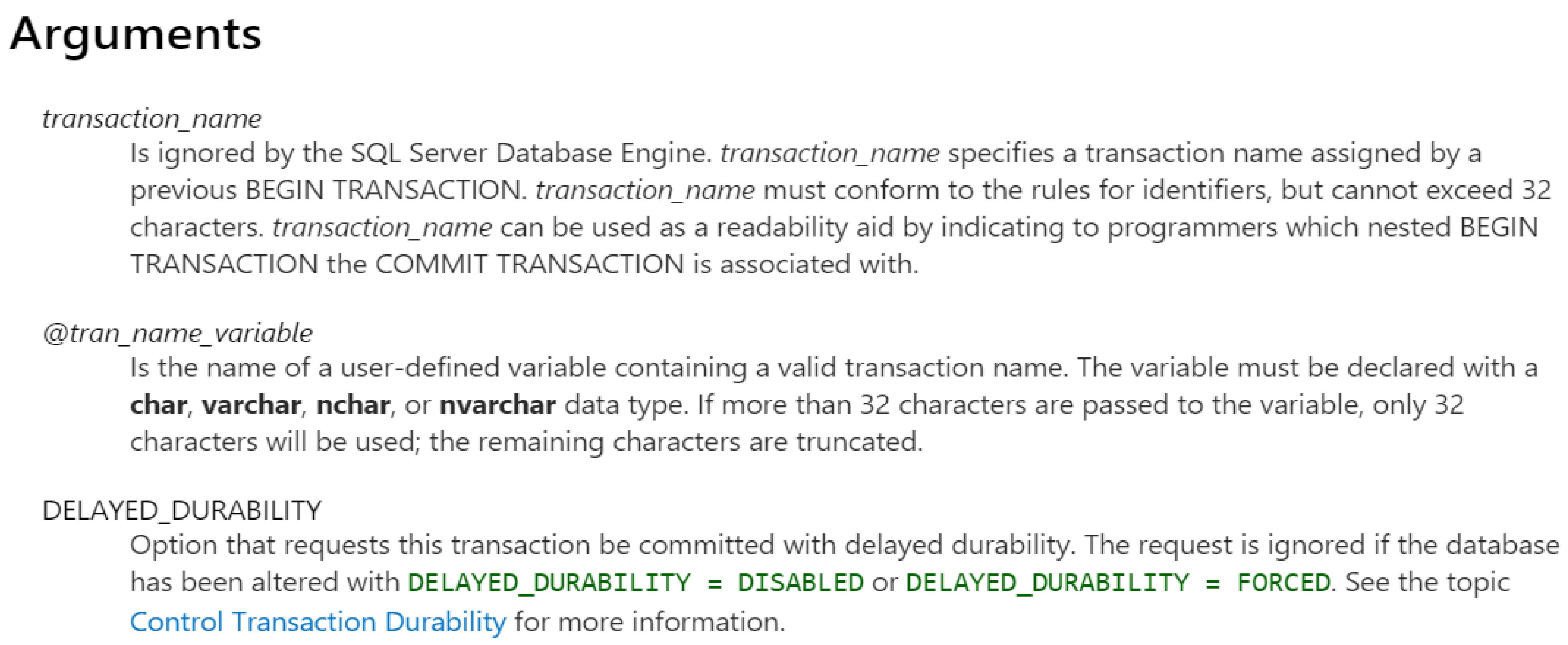
- Within a transaction, duplicate savepoint names are allowed, but a ROLLBACK TRANSACTION using the duplicate savepoint name rolls back only to the most recent SAVE TRANSACTION using that savepoint name.

Commit Transaction

The COMMIT Command:

- The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.
- The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.
- It is the responsibility of the Transact-SQL programmer to issue COMMIT TRANSACTION only at a point when all data referenced by the transaction is logically correct.
- If the transaction committed was a Transact-SQL distributed transaction, COMMIT TRANSACTION triggers MS DTC to use a two-phase commit protocol to commit all of the servers involved in the transaction. If a local transaction spans two or more databases on the same instance of the Database Engine, the instance uses an internal two-phase commit to commit all of the databases involved in the transaction.
- When used in nested transactions, commits of the inner transactions do not free resources or make their modifications permanent. The data modifications are made permanent and resources freed only when the outer transaction is committed. Each COMMIT TRANSACTION issued when @@TRANCOUNT is greater than 1 simply decrements @@TRANCOUNT by 1. When @@TRANCOUNT is finally decremented to 0, the entire outer transaction is committed. Because *transaction_name* is ignored by the Database Engine, issuing a COMMIT TRANSACTION referencing the name of an outer transaction when there are outstanding inner transactions only decrements @@TRANCOUNT by 1.

```
COMMIT [ { TRAN | TRANSACTION } [ transaction_name |
@tran_name_variable ] ]
[ WITH ( DELAYED_DURABILITY = { OFF | ON } ) ] [ ; ]
```



Arguments

transaction_name
Is ignored by the SQL Server Database Engine. *transaction_name* specifies a transaction name assigned by a previous BEGIN TRANSACTION. *transaction_name* must conform to the rules for identifiers, but cannot exceed 32 characters. *transaction_name* can be used as a readability aid by indicating to programmers which nested BEGIN TRANSACTION the COMMIT TRANSACTION is associated with.

@tran_name_variable
Is the name of a user-defined variable containing a valid transaction name. The variable must be declared with a **char**, **varchar**, **nchar**, or **nvarchar** data type. If more than 32 characters are passed to the variable, only 32 characters will be used; the remaining characters are truncated.

DELAYED_DURABILITY
Option that requests this transaction be committed with delayed durability. The request is ignored if the database has been altered with `DELAYED_DURABILITY = DISABLED` or `DELAYED_DURABILITY = FORCED`. See the topic Control Transaction Durability for more information.

SAVEPOINT

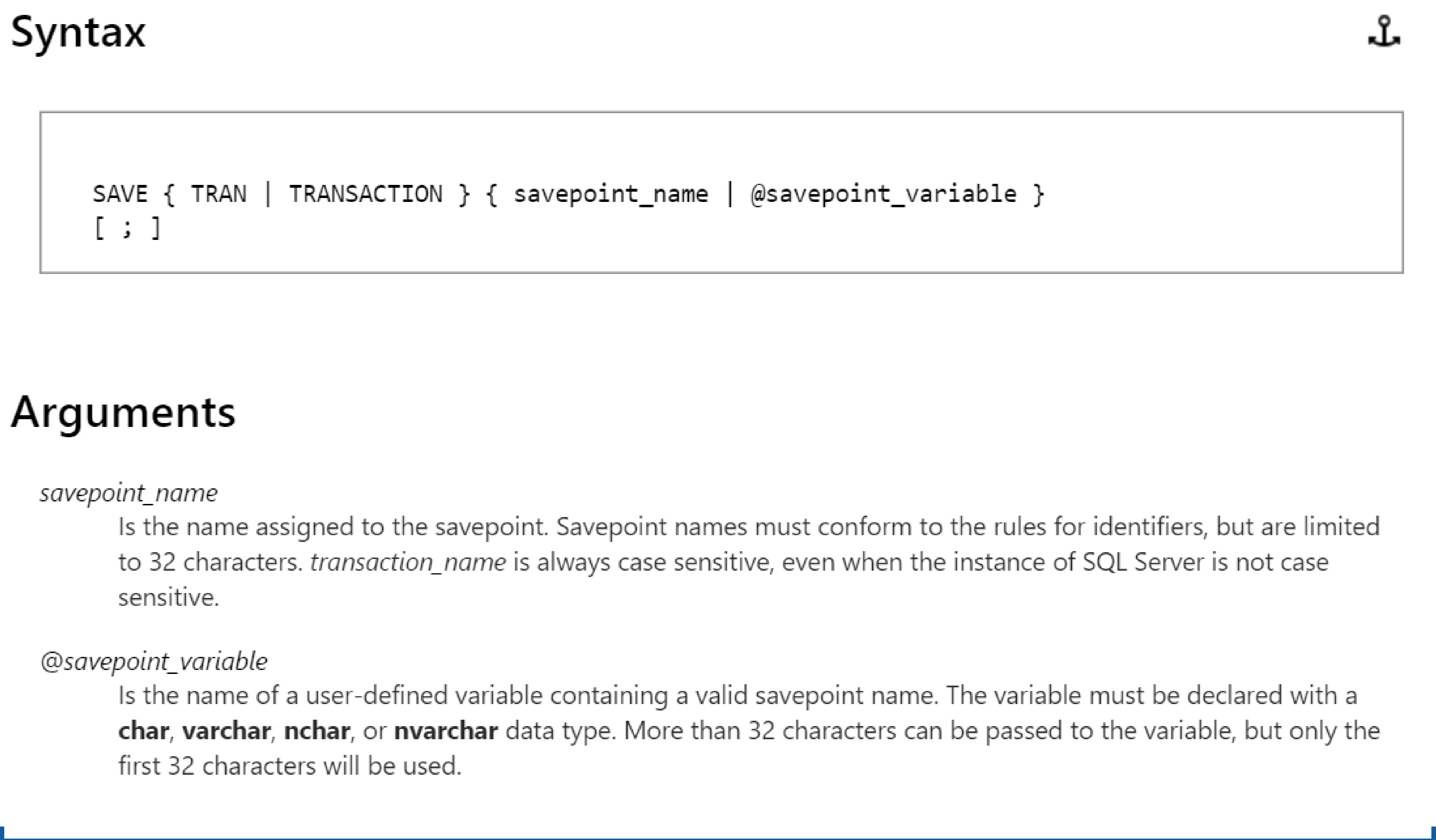
- A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

- A user can set a savepoint, or marker, within a transaction. The savepoint defines a location to which a transaction can return if part of the transaction is conditionally canceled. If a transaction is rolled back to a savepoint, it must proceed to completion with more Transact-SQL statements if needed and a COMMIT TRANSACTION statement, or it must be canceled altogether by rolling the transaction back to its beginning. To cancel an entire transaction, use the form ROLLBACK TRANSACTION *transaction_name*. All the statements or procedures of the transaction are undone.

- Duplicate savepoint names are allowed in a transaction, but a ROLLBACK TRANSACTION statement that specifies the savepoint name will only roll the transaction back to the most recent SAVE TRANSACTION using that name.

- SAVE TRANSACTION is not supported in distributed transactions started either explicitly with BEGIN DISTRIBUTED TRANSACTION or escalated from a local transaction.

SavePoint



Syntax

```
SAVE { TRAN | TRANSACTION } { savepoint_name | @savepoint_variable }
[ ; ]
```

Arguments

savepoint_name
Is the name assigned to the savepoint. Savepoint names must conform to the rules for identifiers, but are limited to 32 characters. *transaction_name* is always case sensitive, even when the instance of SQL Server is not case sensitive.

@savepoint_variable
Is the name of a user-defined variable containing a valid savepoint name. The variable must be declared with a **char**, **varchar**, **nchar**, or **nvarchar** data type. More than 32 characters can be passed to the variable, but only the first 32 characters will be used.

Error Handling

- Implements error handling for Transact-SQL that is similar to the exception handling in the Microsoft Visual C# and Microsoft Visual C++ languages. A group of Transact-SQL statements can be enclosed in a TRY block. If an error occurs in the TRY block, control is passed to another group of statements that is enclosed in a CATCH block.
- A TRY…CATCH construct catches all execution errors that have a severity higher than 10 that do not close the database connection.
- A TRY block must be immediately followed by an associated CATCH block. Including any other statements between the END TRY and BEGIN CATCH statements generates a syntax error.
- A TRY…CATCH construct cannot span multiple batches. A TRY…CATCH construct cannot span multiple blocks of Transact-SQL statements. For example, a TRY…CATCH construct cannot span two BEGIN…END blocks of Transact-SQL statements and cannot span an IF…ELSE construct.

Error Handling Continue..

If there are no errors in the code that is enclosed in a TRY block, when the last statement in the TRY block has finished running, control passes to the statement immediately after the associated END CATCH statement. If there is an error in the code that is enclosed in a TRY block, control passes to the first statement in the associated CATCH block. If the END CATCH statement is the last statement in a stored procedure or trigger, control is passed back to the statement that called the stored procedure or fired the trigger.

When the code in the CATCH block finishes, control passes to the statement immediately after the END CATCH statement. Errors trapped by a CATCH block are not returned to the calling application. If any part of the error information must be returned to the application, the code in the CATCH block must do so by using mechanisms such as SELECT result sets or the RAISERROR and PRINT statements.

TRY…CATCH constructs can be nested. Either a TRY block or a CATCH block can contain nested TRY…CATCH constructs. For example, a CATCH block can contain an embedded TRY…CATCH construct to handle errors encountered by the CATCH code.

GOTO statements cannot be used to enter a TRY or CATCH block. GOTO statements can be used to jump to a label inside the same TRY or CATCH block or to leave a TRY or CATCH block.

The TRY…CATCH construct cannot be used in a user-defined function.

Transact-SQL

```
THROW 51000, 'The record does not exist.', 1;
```

Transact-SQL

```
USE tempdb;
GO
CREATE TABLE dbo.TestRethrow
(    ID INT PRIMARY KEY
);
BEGIN TRY
    INSERT dbo.TestRethrow(ID) VALUES(1);
--  Force error 2627, Violation of PRIMARY KEY constraint to be raised.
    INSERT dbo.TestRethrow(ID) VALUES(1);
END TRY
BEGIN CATCH

    PRINT 'In catch block.';
    THROW;
END CATCH;
```

Error Handling Continue..

Common SQL Functions to use to get more information on the Error:

In the scope of a CATCH block, the following system functions can be used to obtain information about the error that caused the CATCH block to be executed:

- ERROR_NUMBER() returns the number of the error.
- ERROR_SEVERITY() returns the severity.
- ERROR_STATE() returns the error state number.
- ERROR_PROCEDURE() returns the name of the stored procedure or trigger where the error occurred.
- ERROR_LINE() returns the line number inside the routine that caused the error.
- ERROR_MESSAGE() returns the complete text of the error message. The text includes the values supplied for any substitutable parameters, such as lengths, object names, or times.

These functions return NULL if they are called outside the scope of the CATCH block. Error information can be retrieved by using these functions from anywhere within the scope of the CATCH block. For example, the following script shows a stored procedure that contains error-handling functions. In the CATCH block of a TRY…CATCH construct, the stored procedure is called and information about the error is returned.

```sql
-- Verify that the stored procedure does not already exist.
IF OBJECT_ID ( 'usp_GetErrorInfo', 'P' ) IS NOT NULL
    DROP PROCEDURE usp_GetErrorInfo;
GO

-- Create procedure to retrieve error information.
CREATE PROCEDURE usp_GetErrorInfo
AS
SELECT
    ERROR_NUMBER() AS ErrorNumber
    ,ERROR_SEVERITY() AS ErrorSeverity
    ,ERROR_STATE() AS ErrorState
    ,ERROR_PROCEDURE() AS ErrorProcedure
    ,ERROR_LINE() AS ErrorLine
    ,ERROR_MESSAGE() AS ErrorMessage;
GO

BEGIN TRY
    -- Generate divide-by-zero error.
    SELECT 1/0;
END TRY
BEGIN CATCH
    -- Execute error retrieval routine.
    EXECUTE usp_GetErrorInfo;
END CATCH;
```

Cursors

- A cursor is a temporary work area created in the system memory when a SQL statement is executed.
- Cursor is a database object used to manipulate the data in a set on row-by-row basis.
- Implicit cursors are automatically created whenever an SQL statement is executed. Whenever a DML statement is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.
- Explicit cursors are programmer defined cursors for gaining more control over the context area(Perform operation on row by row).

Cursors and use of cursors

Declare Parameters

```
DECLARE @ColExpir datetime
DECLARE @ColFallprotec datetime
DECLARE @ColWorkid int
```

Declare Cursor

```
DECLARE @MyCursor CURSOR
SET @MyCursor = CURSOR
FAST_FORWARD
FOR SELECT
Table_Training_Detalis.DateExpires,Ta
ble_Training_Detalis.Worker_ID
FROM   Table_Courses
```

Use

```
OPEN @MyCursor
FETCH NEXT FROM @MyCursor
INTO @ColExpir,@ColWorkid
WHILE @@FETCH_STATUS = 0
BEGIN
update Table_Workers set WHIMIS=
@ColExpir where Worker_ID=@ColWorkid
FETCH NEXT FROM @MyCursor
INTO @ColExpir,@ColWorkid
END
```

Dispose

```
CLOSE @MyCursor
DEALLOCATE @MyCursor
```

Triggers

- A **trigger** is a special kind of stored procedure that automatically executes when an event occurs in the database server. DML **triggers** execute when a user tries to modify data through a data manipulation language (DML) event. DML events are INSERT, UPDATE, or DELETE statements on a table or view. DDL triggers fire in response to a variety of Data Definition Language (DDL) events like in DML.
- Using triggers is quite valid when their use is justified. For example, they have good value in auditing (keeping history of data) without requiring explicit procedural code with every CRUD command on every table.
- Triggers give you control just before data is changed and just after the data is changed. This allows for:
    - Auditing as mentioned before
    - Validation and business security checking if so is desired. Because of this type of control, you can do tasks such as column formatting before and after inserts into database.

Triggers

- Syntax

```
CREATE TRIGGER keepRecord
ON tbl_test
AFTER INSERT, UPDATE, DELETE
AS
insert into tbl_newRecord (regDate) values (GETDATE());
GO
```

- In this trigger after you insert, update or delete the table (tbl_test) the trigger will insert the date and time on the tbl_newRecord.

Triggers(Instead Of)

INSTEAD OF triggers override the standard actions of the triggering statement: an INSERT, UPDATE, or DELETE. An INSTEAD OF trigger can be defined to perform error or value checking on one or more columns, and then perform additional actions before inserting the record. For example, when the value being updated in an hourly wage column in a payroll table exceeds a specified value, a trigger can be defined to either produce an error message and roll back the transaction, or insert a new record into an audit trail before inserting the record into the payroll table.

An INSTEAD OF trigger can take actions such as:

- Ignoring parts of a batch.
- Not processing a part of a batch and logging the problem rows.
- Taking an alternative action when an error condition is encountered.

Instead Of Trigger can be combined with either Insert(“Instead of Insert”), Update(“Instead of Update”) or Delete(“Instead of Delete”).

Triggers(Instead Of Insert)

- Syntax

```
CREATE TRIGGER InsteadTrigger on InsteadView
INSTEAD OF INSERT
AS
BEGIN
  --Build an INSERT statement ignoring inserted.ID and
  --inserted.ComputedCol.
  INSERT INTO BaseTable
       SELECT Color, Material
       FROM inserted
END;
GO
```

Triggers Vs. Stored procedure

- We can execute a stored procedure whenever we want with the help of the exec command, but a trigger can only be executed whenever an event (insert, delete, and update) is fired on the table on which the trigger is defined.
- We can call a stored procedure from inside another stored procedure but we can't directly call another trigger within a trigger. We can only achieve nesting of triggers in which the action (insert, delete, and update) defined within a trigger can initiate execution of another trigger defined on the same table or a different table.
- Stored procedures can be scheduled through a job to execute on a predefined time, but we can't schedule a trigger.
- Stored procedure can take input parameters, but we can't pass parameters as input to a trigger.

Triggers Vs. Stored procedure

- Stored procedures can return values but a trigger cannot return a value.
- We can use Print commands inside a stored procedure for debugging purposes but we can't use print commands inside a trigger.
- We can use transaction statements like begin transaction, commit transaction, and rollback inside a stored procedure but we can't use transaction statements inside a trigger.
- We can call a stored procedure from the front end (.asp files, .aspx files, .ascx files, etc.) but we can't call a trigger from these files.
- Stored procedures are used for performing tasks. Stored procedures are normally used for performing user specified tasks. They can have parameters and return multiple results sets.
- The Triggers for auditing work: Triggers normally are used for auditing work. They can be used to trace the activities of table events.

Dynamic SQL

- **Dynamic SQL** is a programming technique that enables you to build **SQL** statements **dynamically** at runtime. You can create more general purpose, flexible applications by using **dynamic SQL** because the full text of a **SQL** statement may be unknown at compilation.

- A small piece of the query where the you build your query dynamically.
  - If @CustName Is Not Null
  - Set @SQLQuery = @SQLQuery + ' And (CUST.CustName = @CustName)‘

  - @SQLQuery is the query and is being build at runtime

Dynamic SQL

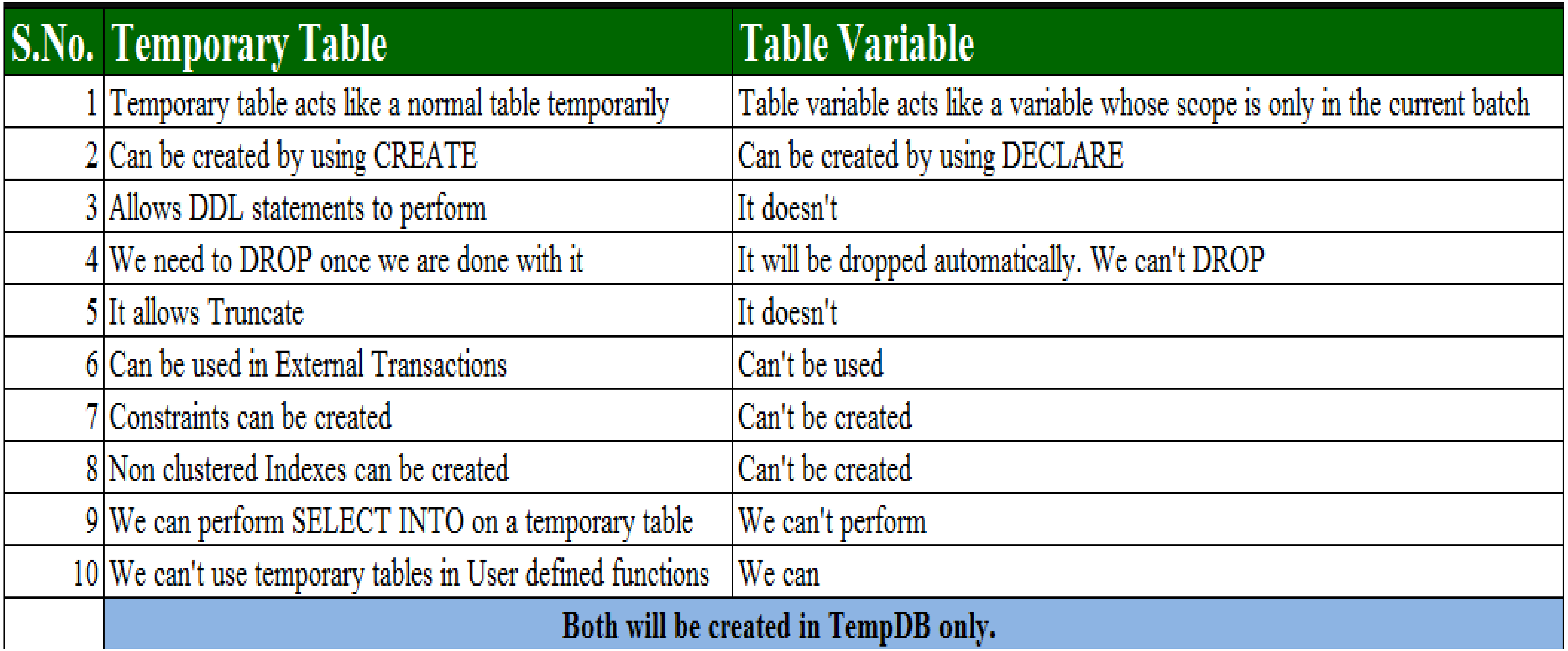
```
CREATE PROCEDURE GetSalesOrders
( @EmployeeName nvarchar(100) = NULL, @Department nvarchar(100) = NULL)
AS SET NOCOUNT ON;

Set @SQLQuery = 'Select * From tblEmployees where (1=1) ‘

If @EmployeeName Is Not Null
Set @SQLQuery = @SQLQuery + ' And (EmployeeName = @EmployeeName)‘

If @Department Is Not Null
Set @SQLQuery = @SQLQuery + ' And (Department = @Department)'

Execute sp_Executesql    @SQLQuery, @EmployeeName, @Department
```

Temp Variable and Temp Table

- Temporary Tables are a great feature that lets you store and process intermediate results by using the same selection, update, and join capabilities that you can use with typical SQL Server tables.
- The temporary tables could be very useful in some cases to keep temporary data. The most important thing that should be known for temporary tables is that they will be deleted when the current client session terminates.

- This is how you declare a local temp Table
  - CREATE TABLE #Temp1 (Name Char(30), seqid integer);
- ## a global temp Table
- The same concept apply to variables where @ is a local variable and @@ global variable

Table Variables and Temp Tables(Differences)



S.No.	Temporary Table	Table Variable
1	Temporary table acts like a normal table temporarily	Table variable acts like a variable whose scope is only in the current batch
2	Can be created by using CREATE	Can be created by using DECLARE
3	Allows DDL statements to perform	It doesn't
4	We need to DROP once we are done with it	It will be dropped automatically. We can't DROP
5	It allows Truncate	It doesn't
6	Can be used in External Transactions	Can't be used
7	Constraints can be created	Can't be created
8	Non clustered Indexes can be created	Can't be created
9	We can perform SELECT INTO on a temporary table	We can't perform
10	We can't use temporary tables in User defined functions	We can
	Both will be created in TempDB only.	

Bulk Data Insertion using XML

- Bulk Data insertion methods using XML can be used when trying to insert more than one row of data in the database.
- Calling Front end application needs to convert all the rows in the XML and then needs to send to SQL Server using a parameter in the stored procedure.
- Common examples are invoice header and invoice details(one or more than one row), order header and order details etc.

Bulk Data Insertion using XML

```
DECLARE @XML AS XML, @hDoc AS INT, @SQL NVARCHAR (MAX)
SELECT @XML = XMLData FROM XMLwithOpenXML

EXEC sp_xml_preparedocument @hDoc OUTPUT, @XML

SELECT CustomerID, CustomerName, Address
FROM OPENXML(@hDoc, 'ROOT/Customers/Customer')
WITH
(
CustomerID [varchar](50) '@CustomerID',
CustomerName [varchar](100) '@CustomerName',
Address [varchar](100) 'Address'
)
EXEC sp_xml_removedocument @hDoc
GO
```

This XML may come from the frontend application

Query execution plan

- A **query plan** (or **query execution plan**) is an ordered set of steps used to access data in a SQL relational database management system. This is a specific case of the relational model concept of access **plans**.
- **Estimated Execution Plan**: When estimated execution plans are generated, the Transact-SQL queries or batches do not execute. Instead, the execution plan that is generated displays the query execution plan that SQL Server Database Engine would most probably use if the queries were actually executed. On the **Query** menu, click **Display Estimated Execution Plan** or click the **Display Estimated Execution Plan** toolbar button.
- **Actual Execution Plan**: When actual execution plans are generated, the Transact-SQL queries or batches execute. The execution plan that is generated displays the actual query execution plan that the SQL Server Database Engine uses to execute the queries. On the **Query** menu, click **Include Actual Execution Plan** or click the **Include Actual Execution Plan** toolbar button

A good article can be found here:

https://www.simple-talk.com/sql/performance/execution-plan-basics/

Database backup and restore

- 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.
- 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 AlwaysOn Availability Groups or database mirroring, and archiving.

Database backup and restore

- 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)
- 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.

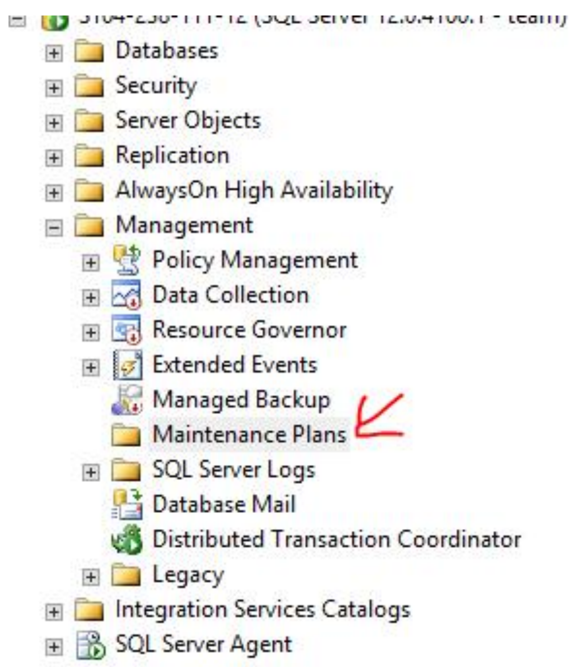
Backup Database Script

```
USE AdventureWorks2012;
GO
BACKUP DATABASE AdventureWorks2012
TO DISK = 'Z:\SQLServerBackups\AdventureWorks2012.Bak'
   WITH FORMAT,
      MEDIANAME = 'Z_SQLServerBackups',
      NAME = 'Full Backup of AdventureWorks2012';
GO
```

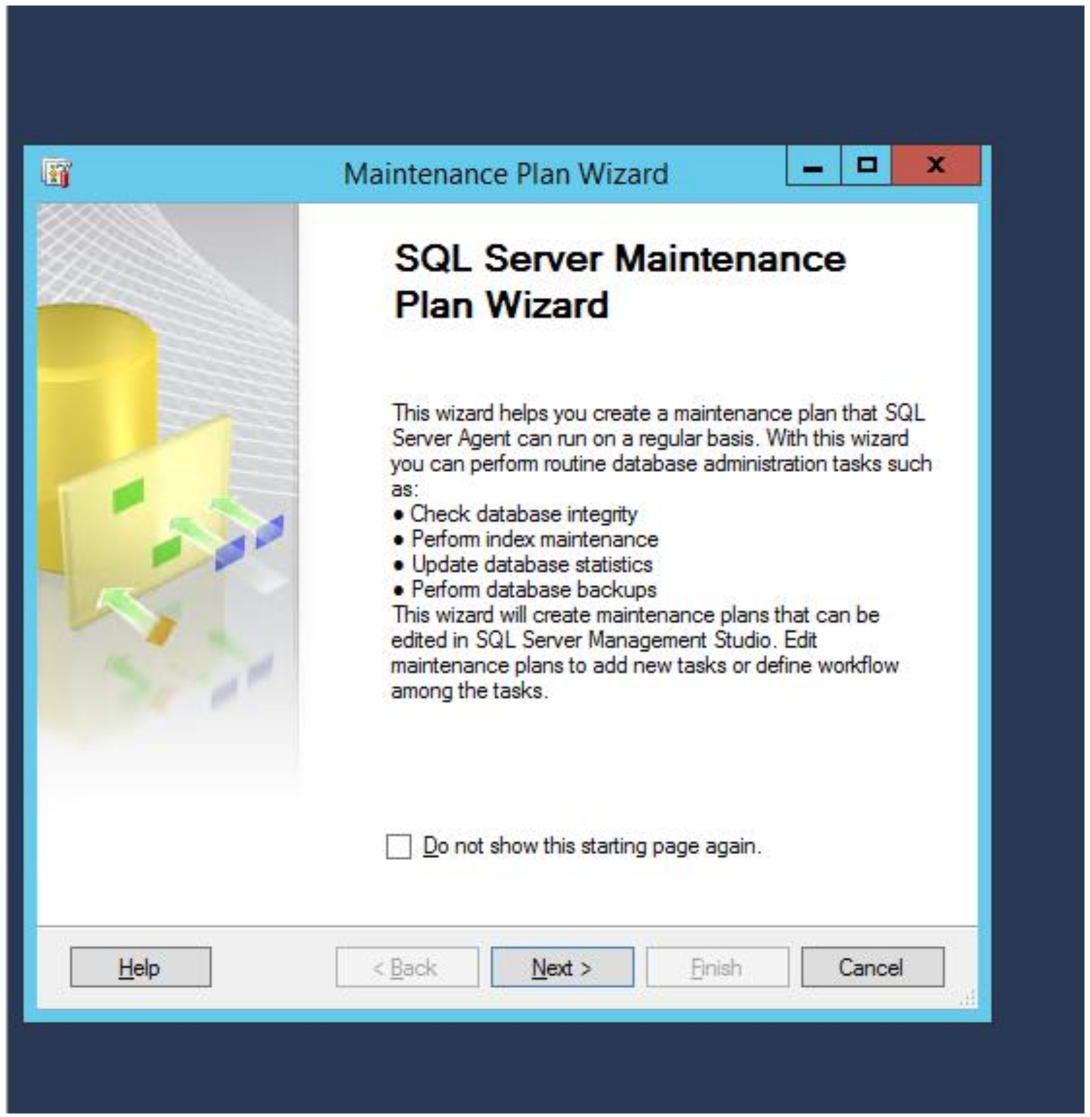
Restore Database Script

```
 USE master;
GO
-- First determine the number and names of the files in the backup.
-- AdventureWorks2012_Backup is the name of the backup device.
RESTORE FILELISTONLY
   FROM AdventureWorks2012_Backup;
-- Restore the files for MyAdvWorks.
RESTORE DATABASE MyAdvWorks
   FROM AdventureWorks2012_Backup
   WITH RECOVERY,
   MOVE 'AdventureWorks2012_Data' TO 'D:\MyData\MyAdvWorks_Data.mdf',
   MOVE 'AdventureWorks2012_Log' TO 'F:\MyLog\MyAdvWorks_Log.ldf';
GO
```

Database backup using Maintenance Plans



- Databases
- Security
- Server Objects
- Replication
- AlwaysOn High Availability
- Management
  - Policy Management
  - Data Collection
  - Resource Governor
  - Extended Events
  - Managed Backup
  - Maintenance Plans
  - SQL Server Logs
  - Database Mail
  - Distributed Transaction Coordinator
  - Legacy
- Integration Services Catalogs
- SQL Server Agent



SQL Server Maintenance Plan Wizard

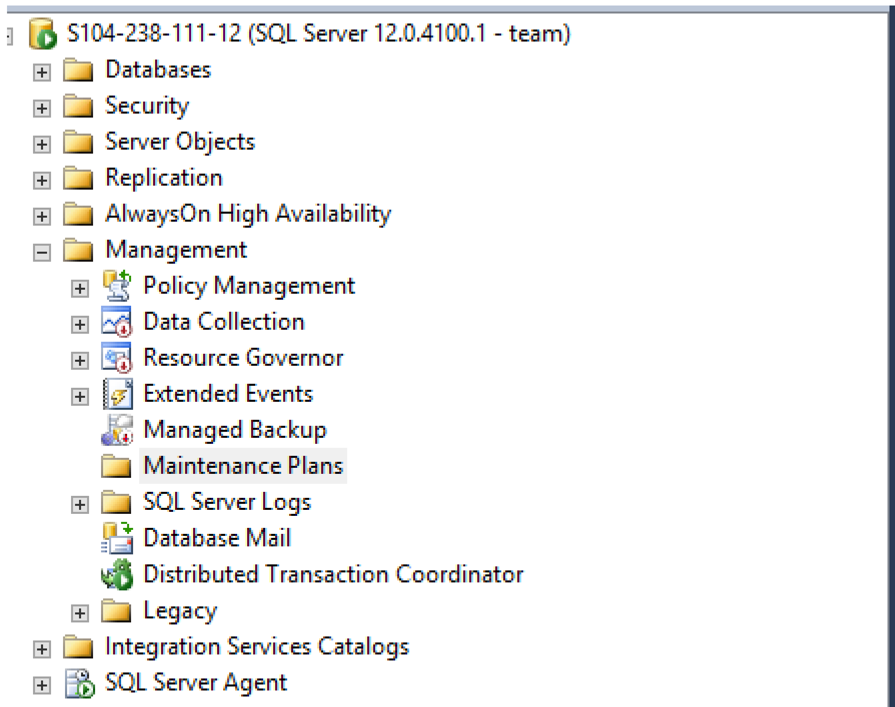
This wizard helps you create a maintenance plan that SQL Server Agent can run on a regular basis. With this wizard you can perform routine database administration tasks such as:

- Check database integrity
- Perform index maintenance
- Update database statistics
- Perform database backups

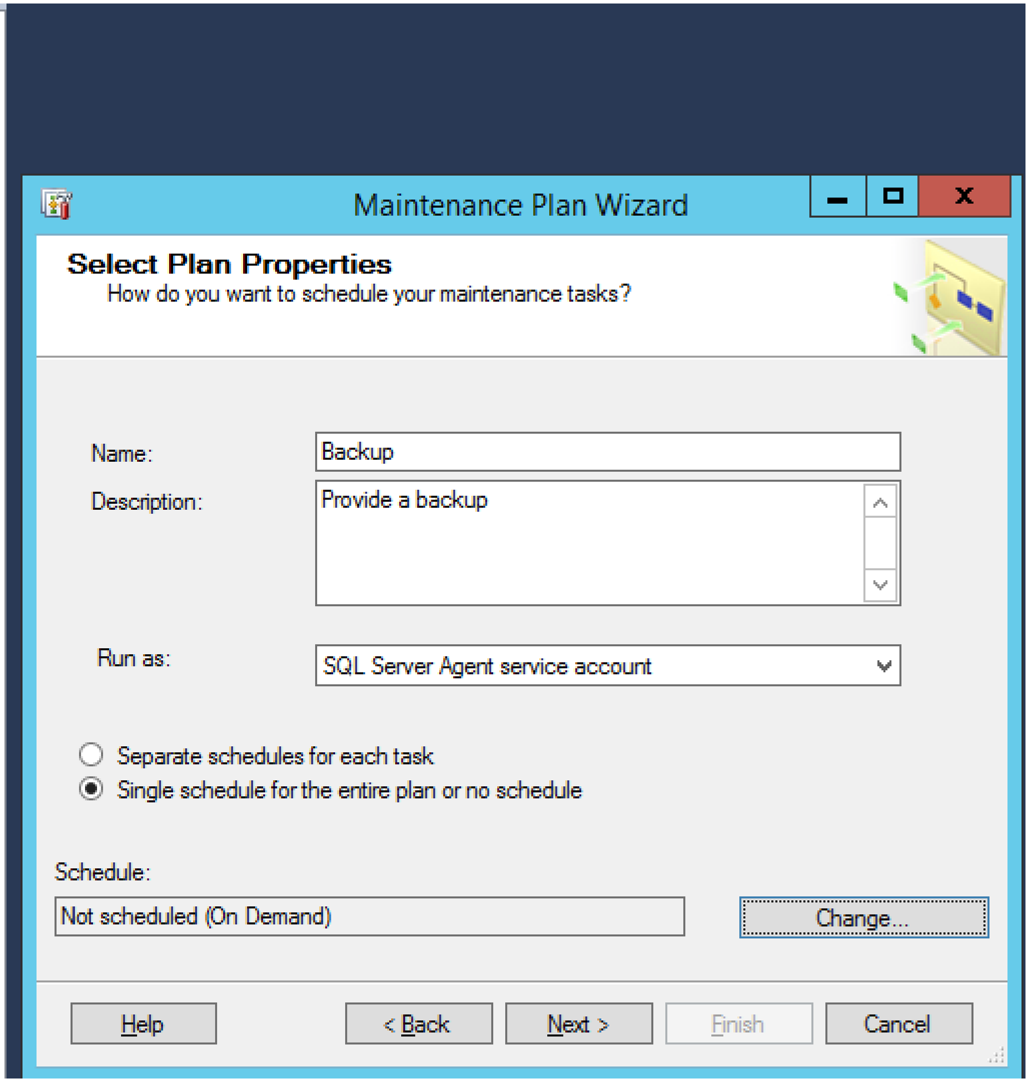
This wizard will create maintenance plans that can be edited in SQL Server Management Studio. Edit maintenance plans to add new tasks or define workflow among the tasks.

☐ Do not show this starting page again.

Database backup using Maintenance Plans



S104-238-111-12 (SQL Server 12.0.4100.1 - team)
- Databases
- Security
- Server Objects
- Replication
- AlwaysOn High Availability
- Management
  - Policy Management
  - Data Collection
  - Resource Governor
  - Extended Events
  - Managed Backup
  - Maintenance Plans
  - SQL Server Logs
  - Database Mail
  - Distributed Transaction Coordinator
  - Legacy
- Integration Services Catalogs
- SQL Server Agent



Maintenance Plan Wizard

Select Plan Properties

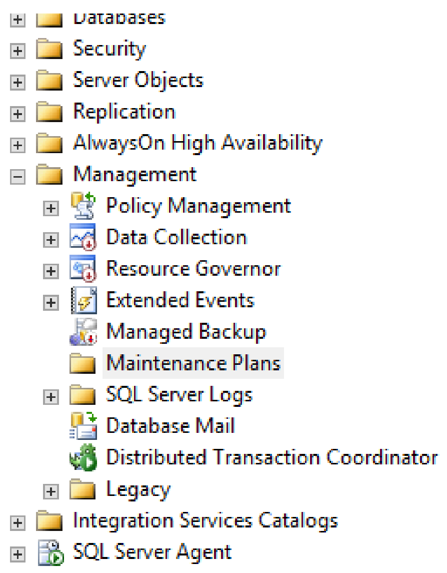
How do you want to schedule your maintenance tasks?

Name:	Backup
Description:	Provide a backup
Run as:	SQL Server Agent service account

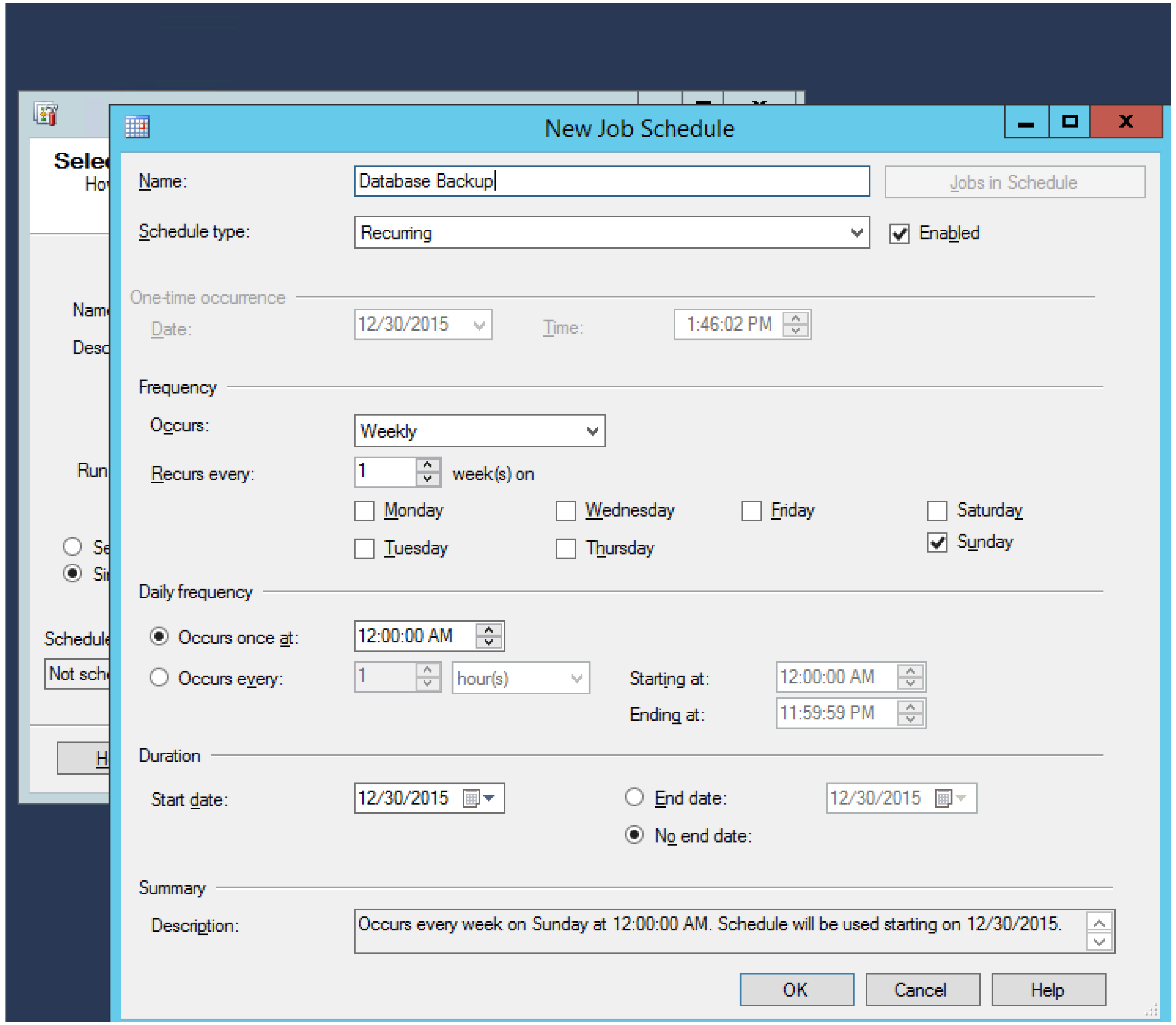
Separate schedules for each task [✓ Single schedule for the entire plan or no schedule]

Schedule: Not scheduled (On Demand)

Database backup using Maintenance Plans



- Databases
- Security
- Server Objects
- Replication
- AlwaysOn High Availability
- Management
  - Policy Management
  - Data Collection
  - Resource Governor
  - Extended Events
  - Managed Backup
  - Maintenance Plans
  - SQL Server Logs
  - Database Mail
  - Distributed Transaction Coordinator
  - Legacy
- Integration Services Catalogs
- SQL Server Agent



New Job Schedule

Name: Database Backup

Schedule type: Recurring   [✓] Enabled

One-time occurrence
Date: 12/30/2015   Time: 1:46:02 PM

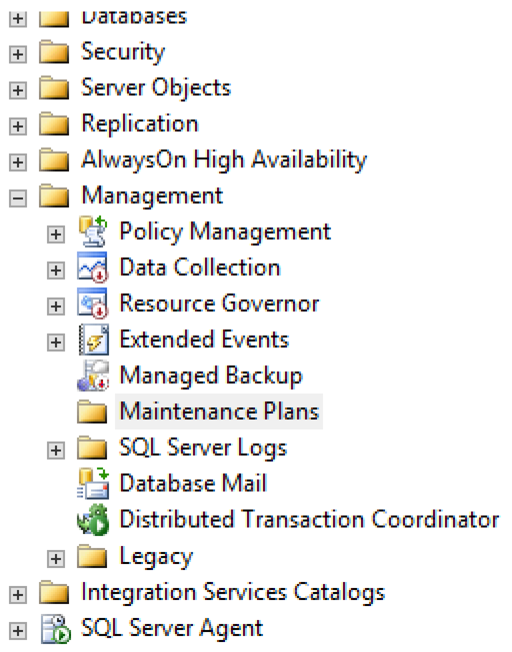
Frequency
Occurs: Weekly
Recurs every: 1 week(s) on
[ ] Monday [ ] Wednesday [ ] Friday [ ] Saturday
[ ] Tuesday [ ] Thursday [✓] Sunday

Daily frequency
(●) Occurs once at: 12:00:00 AM
( ) Occurs every: 1 hour(s)   Starting at: 12:00:00 AM   Ending at: 11:59:59 PM

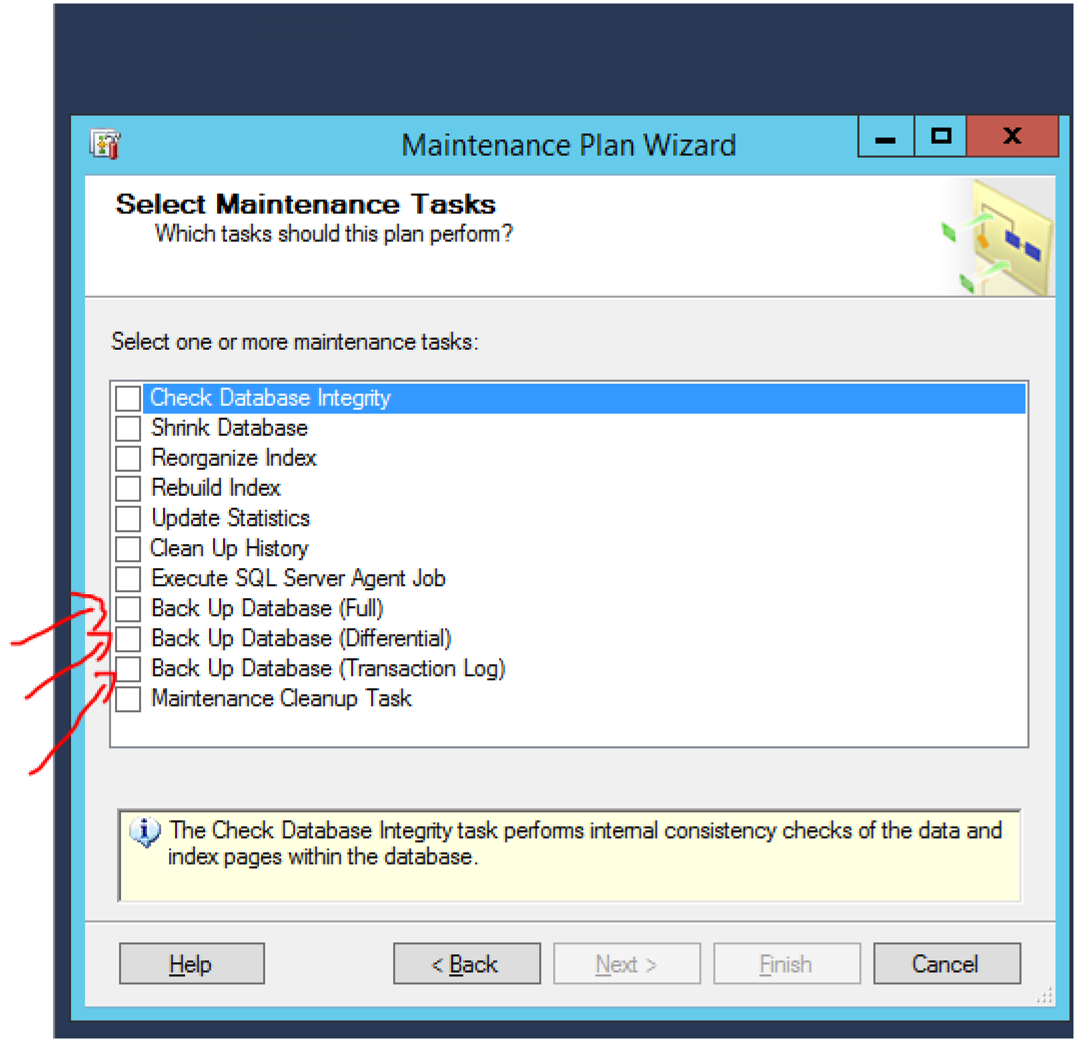
Duration
Start date: 12/30/2015   ( ) End date: 12/30/2015
(●) No end date:

Summary
Description: Occurs every week on Sunday at 12:00:00 AM. Schedule will be used starting on 12/30/2015.

Database backup using Maintenance Plans



Databases
Security
Server Objects
Replication
AlwaysOn High Availability
Management
Policy Management
Data Collection
Resource Governor
Extended Events
Managed Backup
Maintenance Plans
SQL Server Logs
Database Mail
Distributed Transaction Coordinator
Legacy
Integration Services Catalogs
SQL Server Agent



Maintenance Plan Wizard

Select Maintenance Tasks
Which tasks should this plan perform?

Select one or more maintenance tasks:

- [ ] Check Database Integrity
- [ ] Shrink Database
- [ ] Reorganize Index
- [ ] Rebuild Index
- [ ] Update Statistics
- [ ] Clean Up History
- [ ] Execute SQL Server Agent Job
- [ ] Back Up Database (Full)
- [ ] Back Up Database (Differential)
- [ ] Back Up Database (Transaction Log)
- [ ] Maintenance Cleanup Task

The Check Database Integrity task performs internal consistency checks of the data and index pages within the database.