Data Management Using Microsoft SQL Server

Session: 14

Transactions

Objectives

- Define and describe transactions
- Explain the procedure to implement transactions
- Explain the process of controlling transactions
- Explain the steps to mark a transaction
- Distinguish between implicit and explicit transactions
- Explain isolation levels
- Explain the scope and different types of locks
- Explain transaction management

Introduction

➤ A transaction is
  - a single unit of work.
  - is successful only when all data modifications that are made in a transaction are committed and are saved in the database permanently.

➤ If the transaction is rolled back or cancelled, then it means that the transaction has encountered errors and there are no changes made to the contents of the database.

➤ A transaction can be either committed or rolled back.

Need for Transactions 1-5

There are many circumstances where the users need to make many changes to the data in more than one database tables.

In many cases, the data will be inconsistent that executes the individual commands.

Suppose if the first statement executes correctly but the other statements fail then the data remains in an incorrect state.

A **good scenario will be the funds transfer activity in a banking system which will need an** `INSERT` **and two** `UPDATE` **statements**.

First, the user has to increase the balance of the destination account and then, decrease the balance of the source account.

The user has to check that the transactions are committed and whether the same changes are made to the source account and the destination account.

SQL Server 2012

Defining Transactions: A logical unit of work must exhibit four properties, called the atomicity, consistency, isolation, and durability (ACID) properties, to qualify as a transaction.

Atomicity: If the transaction has many operations then all should be committed.

Consistency: The sequence of operations must be consistent.

Isolation: The operations that are performed must be isolated from the other operations on the same server or on the same database.

Durability: The operations that are performed on the database must be saved and stored in the database permanently.

Implementing Transactions: SQL Server supports transactions in several modes.

➢ **Autocommit Transactions**: Every single-line statement is automatically committed as soon as it completes.

➢ **Explicit Transactions**: Every transaction explicitly starts with the `BEGIN TRANSACTION` statement and ends with a `ROLLBACK` or `COMMIT` transaction.

➢ **Implicit Transactions**: A new transaction is automatically started when the earlier transaction completes and every transaction is explicitly completed by using the `ROLLBACK` or `COMMIT` statement.

➢ **Batch-scoped Transactions**: These transactions are related to Multiple Active Result Sets (MARS).

SQL Server 2012

➢ **Transactions Extending Batches**

Transaction statements identify the block of code that should either fail or succeed and provide the facility where the database engine can undo or roll back the operations.

Users can add error-handling code to roll back the transaction in case of errors. The error-handling code will undo the partial changes that were made before the error had occurred.

Controlling Transactions

Transactions can be controlled through applications by specifying the beginning and ending of a transaction.

This is done by using the database API functions or Transact-SQL statements.

When a transaction is started on a connection, all Transact-SQL statements are executed on the same connection and are a part of the connection until the transaction ends.

Transactions are managed at the connection level.

One of the ways users can start and end transactions is by using Transact-SQL statements.

Explicit transaction mode starts a transaction by using a `BEGIN TRANSACTION` statement.

Users can end a transaction using the `ROLLBACK` or `COMMIT` statements.

`BEGIN TRANSACTION` statement marks the beginning point of an explicit or local transaction.

Syntax:

```
BEGIN { TRAN | TRANSACTION }

[ { transaction_name | @tran_name_variable }

[ WITH MARK [ 'description' ] ]
]
[ ; ]
```

where,

`transaction name`: specifies the name that is assigned to the transaction. It should follow the rules for identifiers and limit the identifiers that are 32 characters long.

`@tran name variable`: specifies the name of a user-defined variable that contains a valid transaction name.

`WITH MARK['description']`: specifies the transaction that is marked in the log. The description string defines the mark.

➢ Following code snippet shows how to create and begin a transaction:

```
USE AdventureWorks2012;
GO
DECLARE @TranName VARCHAR(30);
SELECT @TranName = 'FirstTransaction';
BEGIN TRANSACTION @TranName;
DELETE FROM HumanResources.JobCandidate
WHERE JobCandidateID = 13;
```

SQL Server 2012

COMMIT TRANSACTION statement marks an end of a successful implicit or explicit transaction.

Syntax:

```
COMMIT { TRAN | TRANSACTION } [ transaction_name | @tran_name_variable ] ]
[ ; ]
```

where,

transaction name: specifies the name that is assigned by the previous BEGIN TRANSACTION statement. It should follow the rules for identifiers and do not allow identifiers that are 32 characters long.

@tran name variable: specifies the name of a user-defined variable that contains a valid transaction name. The variable can be declared as char, varchar, nchar, or nvarchar data type. If more than 32 characters are passed to the variable, then only 32 characters are used and the remaining characters will be truncated.

➢ Following code snippet shows how to commit a transaction:

```
BEGIN TRANSACTION;
GO
DELETE FROM HumanResources.JobCandidate
WHERE JobCandidateID = 11;
GO
COMMIT TRANSACTION;
GO
```

COMMIT WORK statement marks the end of a transaction.

Syntax:

```
COMMIT [ WORK ]
[ ; ]
```

COMMIT TRANSACTION and COMMIT WORK are identical except for the fact that COMMIT TRANSACTION accepts a user-defined transaction name.

Marking a Transaction

➢ Following code snippet shows how to mark a transaction:

```
BEGIN TRANSACTION DeleteCandidate
WITH MARK N'Deleting a Job Candidate';
GO
DELETE FROM HumanResources.JobCandidate
WHERE JobCandidateID = 11;
GO
COMMIT TRANSACTION DeleteCandidate;
```

ROLLBACK TRANSACTION - This transaction rolls back or cancels an implicit or explicit transaction to the starting point of the transaction, or to a savepoint in a transaction.

Syntax:

```
COMMIT [ WORK ]
[ ; ]
```

ROLLBACK TRANSACTION - This transaction rolls back or cancels an implicit or explicit transaction to the starting point of the transaction, or to a savepoint in a transaction.

Syntax:

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

where,

transaction_name: specifies the name that is assigned to the BEGIN TRANSACTION statement. It should confirm the rules for identifiers.

@tran_name_variable: specifies the name of a user-defined variable that contains a valid transaction name. The variable can be declared as char, varchar, nchar, or nvarchar data type.

savepoint_name: specifies the savepoint_name from a SAVE TRANSACTION statement. Use savepoint_name only when a conditional rollback affects a part of a transaction.

@savepoint_variable: specifies the name of savepoint variable that contain a valid savepoint name. The variable can be declared as char, varchar, nchar, or nvarchar data type.

➢ Following code snippet demonstrates the use of ROLLBACK:

```
USE Sterling;
GO
CREATE TABLE ValueTable ([value] char)
GO
```

➢ Following code snippet shows a transaction that inserts two records into ValueTable:

```
BEGIN TRANSACTION
INSERT INTO ValueTable VALUES('A');
INSERT INTO ValueTable VALUES('B');
GO
ROLLBACK TRANSACTION
INSERT INTO ValueTable VALUES('C');
SELECT [value] FROM ValueTable;
```
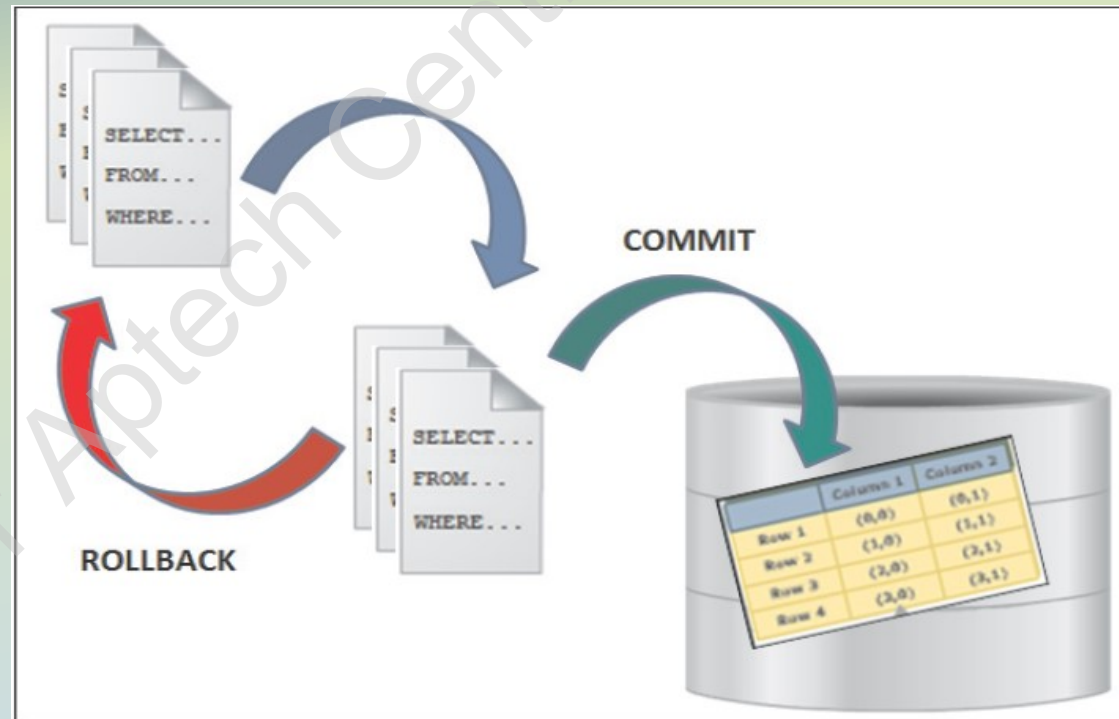
➢ Then, it rolls back the transaction and again inserts one record into ValueTable.

ROLLBACK WORK statement rolls back a user-specified transaction to the beginning of the transaction.

Syntax:

```
ROLLBACK [ WORK ]
[ ; ]
```

SAVE TRANSACTION statement sets a savepoint within a transaction.

Syntax:

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

where,

savepoint_name: specifies the savepoint_name assigned. These names conform to the rules of identifiers and are restricted to 32 characters.

@savepoint_variable: specifies the name of a user-defined variable that contain a valid savepoint name. The variable can be declared as char, varchar, nchar, or nvarchar data type. More than 32 characters are allowed to pass to the variables but only the first 32 characters are used.

➤ Following code snippet demonstrates how to use a savepoint transaction:

```
CREATE PROCEDURE SaveTranExample

@InputCandidateID INT
AS
DECLARE @TranCounter INT;
SET @TranCounter = @@TRANCOUNT;
IF @TranCounter > 0
    SAVE TRANSACTION ProcedureSave;
ELSE
    BEGIN TRANSACTION;
    DELETE HumanResources.JobCandidate
        WHERE JobCandidateID = @InputCandidateID;
    IF @TranCounter = 0
        COMMIT TRANSACTION;
        IF @TranCounter = 1
        ROLLBACK TRANSACTION ProcedureSave;
GO
```

@@TRANCOUNT system function returns a number of BEGIN TRANSACTION statements that occur in the current connection.
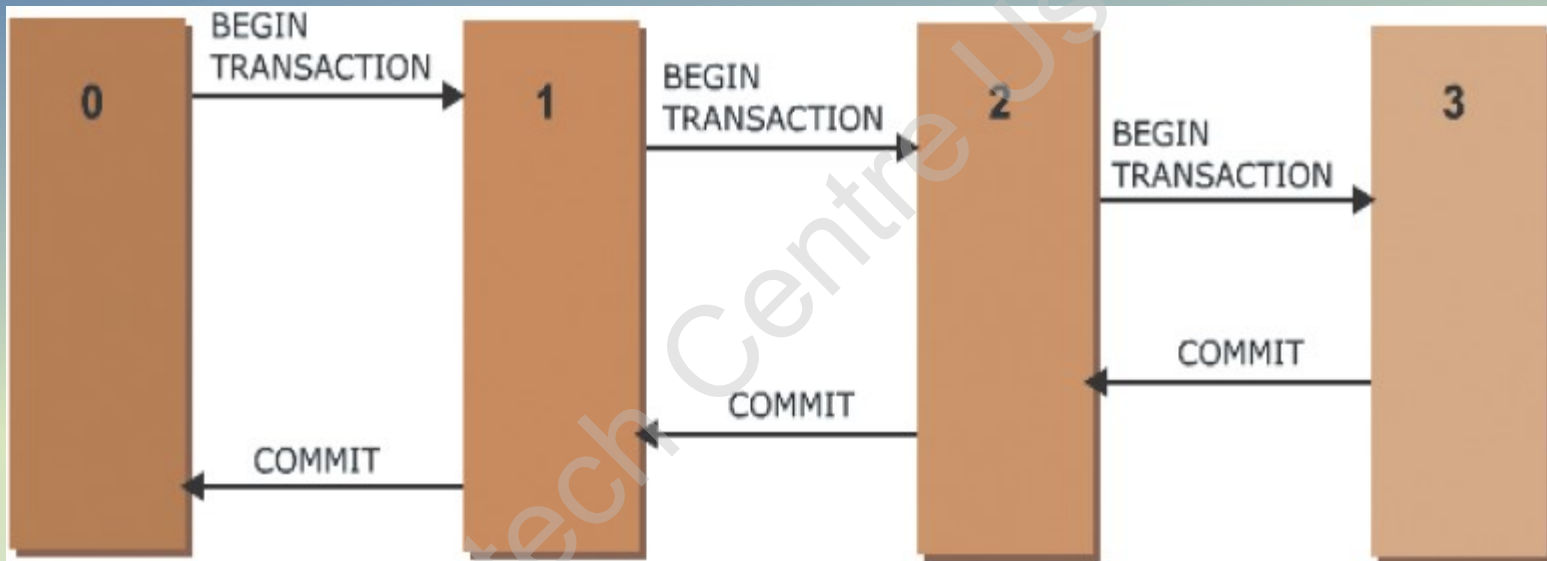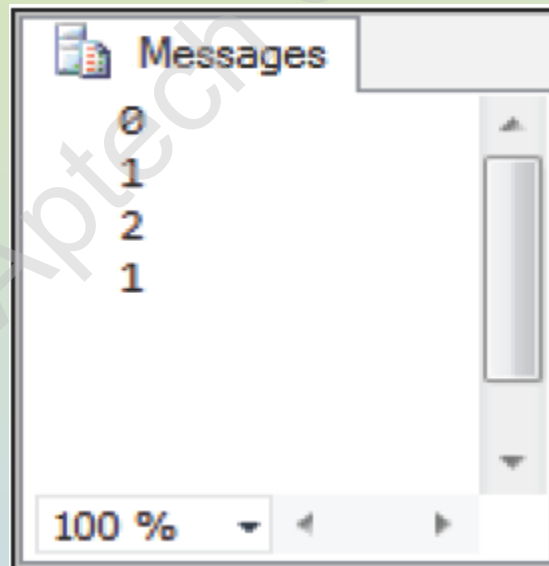

Syntax:

```
@@TRANCOUNT
```

@@TRANCOUNT 2-2

➢ Following code snippet shows the effect that nested BEGIN and COMMIT statements have on the @@TRANCOUNT variable:

```
PRINT @@TRANCOUNT
BEGIN TRAN
  PRINT @@TRANCOUNT
  BEGIN TRAN
    PRINT @@TRANCOUNT
  COMMIT
  PRINT @@TRANCOUNT
COMMIT
PRINT @@TRANCOUNT
```

Output:

Messages
```
0
1
2
1
```

Marking a Transaction 1-3

Users can use transaction marks to recover the related updates made to two or more related databases.

Marking related transactions on a routine basis in every single related database creates a sequence of common recovery points in a database.

In case of any disaster, user can restore each of the databases to the same transaction mark in order to recover them to a consistent point.

Marking a Transaction 2-3

Concerns for Using Marked Transactions

- As the transaction mark consume log space, use them only for transactions that play an important role in the database recovery strategy.

- When the marked transaction is committed, then a row is inserted in the `logmarkhistory` table in `msdb`.

- If a marked transaction spans over multiple databases on different servers or on the same database server, the marks must be logged in the records of all affected databases.

Marking a Transaction 3-3

➢ For creating a marked transaction in a set of databases, the steps to be followed are as follows:
- Name the transaction in the `BEGIN TRANSACTION` statement and use the `WITH MARK` clause
- Execute an update against all of the databases in the set

➢ Following code snippet demonstrates for creating a marked transaction in a set of databases:

```
USE AdventureWorks2012
GO
BEGIN TRANSACTION ListPriceUpdate
  WITH MARK 'UPDATE Product list prices';
GO
UPDATE Production.Product
  SET ListPrice = ListPrice * 1.20
  WHERE ProductNumber LIKE 'BK-%';
GO
COMMIT TRANSACTION ListPriceUpdate;
GO
```

➢ Following table lists the differences between implicit and explicit transactions:

Implicit	Explicit
These transactions are maintained by SQL Server for each and every DML and DDL statements	These transactions are defined by programmers
These DML and DDL statements execute under the implicit transactions	DML statements are included to execute as a unit
SQL Server will roll back the entire statement	SELECT Statements are not included as they do not modify data

Isolation Levels 1-2

Transactions identify the isolation levels that define the degree to which one transaction must be isolated from the data modifications or resource that are made by the other transactions.

When one transaction changes a value and a second transaction reads the same value before the original change has been committed or rolled back, it is called as a dirty read.

A lower isolation level increases the capability of several users to access data at the same time. A higher isolation level decreases the types of concurrency effects which user may encounter.

Isolation Levels 2-2

➤ Following table lists the concurrency effects that are allowed by the different isolation levels:

Isolation Level	Dirty Read	NonRepeatable Read
Read committed	No	Yes
Read uncommitted	Yes	No
Snapshot	No	No
Repeatable Read	No	No
Serializable	No	No

SQL Server Database Engine locks the resources that use different lock modes, which determine the resources that are accessible to concurrent transactions.

➢ Following table lists the lock modes used by the Database Engine:

Lock Mode	Description
Update	Is used on resources that are to be updated.
Shared	Is used for read operations that do not change data such as SELECT statement.
Intent	Is used to establish a hierarchy of locks.
Exclusive	Is used for INSERT, UPDATE, or DELETE data-modification operations.
BULK UPDATE	Is used while copying bulk data into the table.
Schema	Is used when the operation is dependent on the table schema.

Scope and Different Types of Locks 2-6

Update Locks

- avoid common forms of deadlock
- When two transactions acquire a shared lock on a resource and try to update data simultaneously, the same transaction attempts the lock conversion to an exclusive lock
- Only one transaction can obtain an update lock to a resource at a time
- When a transaction modifies a resource, then the update lock is converted to an exclusive lock

Shared Locks

- allow parallel transactions to read a resource under pessimistic concurrency control
- Transactions can change the data while shared locks exist on the resource
- Shared locks are released on a resource once the read operation is completed

Scope and Different Types of Locks 3-6

Exclusive Locks

- prevent access to resources by concurrent transactions
- no other transaction can change data and read operations take place only through the read uncommitted isolation level or `NOLOCK` hint
- DML statements usually request both exclusive and shared locks
- When a transaction modifies a resource, then the update lock is converted to an exclusive lock

Intent Locks

- used for protecting and places an exclusive or shared lock on the resource that is at a lower level in the lock hierarchy
- are acquired before a lock at the low level and hence, indicate intent to place locks at low level
- useful for two purposes:
  - prevent other transactions from changing the higher-level resource
  - improve the efficiency of the Database Engine for identifying the lock conflicts

Setting the intent lock at the table level protects other transaction from subsequently acquiring an exclusive lock on the table containing pages.

➢ Following table lists the lock modes in Intent locks:

Lock Mode	Description
Intent shared (IS)	Protects the requested shared lock on some resources that are lower in the hierarchy.
Intent exclusive (IX)	Protects the requested exclusive lock on some resources lower in the hierarchy. IX is a superset of IS, that protects requesting shared locks on lower level resources.
Shared with Intent Exclusive (SIX)	Protects the requested shared lock on all resources lower in the hierarchy and intent exclusive locks on some of the lower level resources. Concurrent IS locks are allowed at the top-level resource.
Intent Update (IU)	Protects the requested update locks on all resources lower in the hierarchy. IU locks are used only on page resources. IU locks are converted to IX locks if an update operation takes place.
Shared intent update (SIU)	Provides the combination of S and IU locks, as a result of acquiring these locks separately and simultaneously holding both locks.
Update intent exclusive (UIX)	Provides the combination of U and IX locks, as a result of acquiring these locks separately and simultaneously holding both locks.

Scope and Different Types of Locks 5-6

➢ Many persons are involved in the design, use, and maintenance of a large database with a few hundred users.

Bulk Update Locks

- are used by the database engine
- are used when a large amount of data is copied into a table (bulk copy operations) and either the table lock on bulk load option is set or the `TABLOCK` hint is specified using the `sp_table` option
- allow multiple threads to load bulk data continuously in the same table

Schema Locks

- are used by Database Engine while performing a table DDL operation such as dropping a table or a column
- prevents concurrent access to the table, which means a schema lock blocks all external operations until the lock releases
- are used by the database engine while compiling and executing the queries

SQL Server 2012

Key-Range Locks

- protect a collection of rows that are implicitly present in a recordset which is being read by a Transact-SQL statement while using the serializable transaction isolation level
- prevent the phantom deletions or insertions in the recordset that accesses a transaction

Transaction Management

Every single statement that is executed is, by default, transactional in SQL Server.

When the users use explicit BEGIN TRAN/COMMIT TRAN commands, they can group them together as an explicit transaction.

However, data stored in this form is not permanent. Records in such manual files can only be maintained for a few months or few years.

Transaction Log 1-3

SQL Server database has a transaction log, which records all transactions and the database modifications made by every transaction.

Operations supported by the transaction log are as follows:

- ➢ Individual transactions recovery
- ➢ Incomplete transactions recovery when SQL Server starts
- ➢ Transactional replication support
- ➢ Disaster recovery solutions and high availability support
- ➢ Roll back a file, restored database, filegroup, or page forward to the point of failure

➢ **Truncating a Transaction Log**

Truncating a log frees the space in the log file for reusing the transaction log.

➢ In a simple recovery model after the checkpoint
➢ In a bulk-logged recovery model or full recovery model, if the checkpoint is occurred ever since the last backup, truncation occurs after a log backup

Log truncations are delayed due to many reasons. Users can also discover if anything prevents the log truncation by querying the `log_reuse_wait_desc` and `log_reuse_wait` columns of the `sys.databases` catalog view.

➤ Following table lists the values of some of these columns:

Log_reuse_wait value mode	Log_reuse_wait desc value	Description
0	NOTHING	Specifies that at present, there are more than one reusable virtual log file.
1	CHECKPOINT	Specifies that there is no checkpoint occurred since the last log truncation, or the head of the log has not moved beyond a virtual log file.
2	LOG_BACKUP	Specifies a log backup that is required before the transaction log truncates.
3	ACTIVE_BACKUP_OR_RESTORE	Specifies that the data backup or a restore is in progress.
4	ACTIVE_TRANSACTION	Specifies that a transaction is active.
5	DATABASE_MIRRORING	Specifies that the database mirroring is paused, or under high-performance mode, the mirror database is significantly behind the principal database.

Summary

- A transaction is a sequence of operations that works as a single unit.

- Transactions can be controlled by an application by specifying a beginning and an ending.

- BEGIN TRANSACTION marks the beginning point of an explicit or local transaction.

- COMMIT TRANSACTION marks an end of a successful implicit or explicit transaction.

- ROLLBACK with an optional keyword WORK rolls back a user-specified transaction to the beginning of the transaction.

- @@TRANCOUNT is a system function that returns a number of BEGIN TRANSACTION statements that occur in the current connection.

- Isolation levels are provided by the transaction to describe the extent to which a single transaction needs to be isolated from changes made by other transactions.

- The SQL Server Database Engine locks the resources using different lock modes, which determine the resources that are accessible to concurrent transactions.