**Transactions**

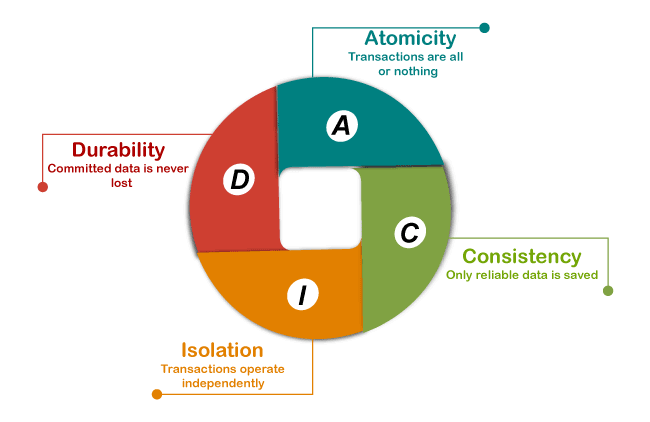
Transactions are units of work that are executed as a single, logical operation. The execution of transaction can be in a manual fashion by a user, or automatically by some sort of a database program.

A transaction is to make some changes to the database; for example, if you are creating, updating or deleting a record from the table, then you are performing a transaction on that table.

Practically, transactions allow you to group one or more SQL statements into a single execution unit, ensuring that either all of the statements are successfully completed or none of them are.

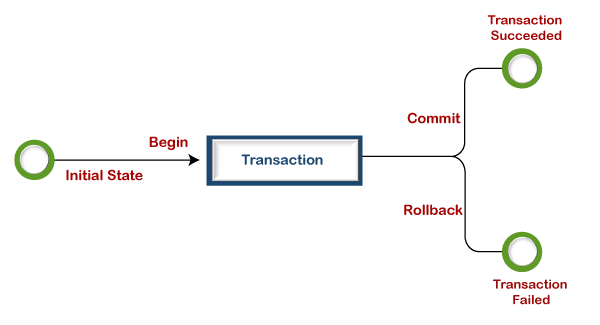
Transactions in SQL Server are typically used to maintain the integrity of the database and ensure that data remains consistent even in the event of errors, crashes, or concurrent access by multiple users. Transactions follow the **principles of ACID** (Atomicity, Consistency, Isolation, Durability) to guarantee reliable database operations:

* **Atomicity**: Transactions are atomic, meaning they are treated as a single, indivisible unit of work. Either all of the operations within the transaction are successfully completed, or none of them are. There is no partial execution.
* **Consistency**: Transactions ensure that the database remains in a consistent state before and after the transaction is executed. This means that the integrity constraints, such as foreign key relationships and unique constraints, are maintained.
* **Isolation**: Transactions are isolated from each other to prevent interference and maintain data integrity in a multi-user environment. Each transaction operates as if it is the only transaction running on the database, regardless of concurrent transactions.
* **Durability**: Once a transaction is committed, the changes made by the transaction are permanent and survive system failures. These changes are stored in non-volatile storage, such as disk, to ensure durability.



In SQL Server, you can begin a transaction using the BEGIN TRANSACTION statement, to make changes to the database, and then either commit the transaction using COMMIT TRANSACTION to make the changes permanent, or roll back the transaction using ROLLBACK TRANSACTION to undo the changes and restore the database to its previous state.

As the below picture shows, a transaction cannot be successful until all of the operations in the set are completed. It means that if any argument fails, the transaction operation will fail. Each transaction begins with the first executable SQL statement and ends when it finds a commit or rollback, either explicitly or implicitly.



Banking database system can be a good example for the concept of a transaction. Suppose a bank customer wants to transfer money from one account to another by using ATM:

* The first step is to check the availability of the requested amount in the account.
* The second step deducts the amount from the first account if the amount is available.
* The third step is to deposit the money to the second account. At this step, transaction is either successful or failed. If successful, database gets updated based on transaction; otherwise, the transaction will be rolled back into previous state.

The basic principle behind transactions is that if one of the statements returns an error, the entire set of changes is rolled back to ensure data integrity. And if the transactions become successful, all changes will be permanent on the database. Hence, if there is a power outage or other issues when withdrawing money from an ATM, transactions guarantee that our balance remains consistent.

Syntax:

BEGIN TRANSACTION;

UPDATE TableName

SET Column1 = Value1

WHERE Condition;

INSERT INTO TableName (Column1, Column2)

VALUES (Value1, Value2);

COMMIT TRANSACTION;

**Transaction modes or types in SQL Server**

1. Implicit Transactions (Auto-Commit):

In this mode, each SQL statement is treated as a separate transaction that is automatically committed after execution if it completes successfully. There's no need to explicitly begin and commit transactions.

Here, each statement is implicitly in its own transaction and automatically committed after execution:

INSERT INTO Employee (EmployeeID, Name, Department, Salary) VALUES (1, 'John Doe', 'IT', 50000.00);

UPDATE Employee SET Salary = 55000.00 WHERE EmployeeID = 1;

DELETE FROM Employee WHERE EmployeeID = 2;

In this mode, each SQL statement is executed in its own transaction and automatically committed if successful. If an error occurs during the execution of a statement, the transaction is automatically rolled back.

2. Explicit Transactions (Manual Commit/Rollback):

In this mode, transactions must be explicitly started using BEGIN TRANSACTION, and then explicitly committed using COMMIT TRANSACTION or rolled back using ROLLBACK TRANSACTION.

BEGIN TRANSACTION;

-- Data modification statements within the explicit transaction

INSERT INTO Employee (EmployeeID, Name, Department, Salary) VALUES (1, 'John Doe', 'IT', 50000.00);

UPDATE Employee SET Salary = 55000.00 WHERE EmployeeID = 1;

-- Check if data is correct

SELECT \* FROM Employee;

-- Commit the transaction

COMMIT TRANSACTION;

In this mode, transactions are explicitly started and can span multiple SQL statements. They must be explicitly committed or rolled back by the user.

Note: Implicit transactions provide convenience but less control, while explicit transactions offer more control but require manual management.

In practical terms, the terms "auto-commit" and "implicit transactions" are often used interchangeably, especially in the context of SQL Server. In both cases:

* Each individual SQL statement is treated as a separate transaction.
* The transaction is automatically committed after the SQL statement executes successfully.
* If an error occurs during the execution of the SQL statement, the transaction is automatically rolled back.

So, in essence, there's no practical difference between auto-commit and implicit transactions in SQL Server. Both terminologies refer to the same behavior where transactions are managed automatically for each SQL statement without the need for explicit transaction management commands like BEGIN TRANSACTION, COMMIT TRANSACTION, or ROLLBACK TRANSACTION.

However, it's worth noting that "implicit transactions" might encompass a broader concept in some contexts beyond just the auto-commit behavior, including settings related to implicit transaction modes. But when discussing transaction behavior specifically in SQL Server, the terms are often used synonymously.

3. Save-point Transactions: These transactions allow defining save-points within a transaction using the SAVE TRANSACTION statement. Save-points provide a way to roll back portions of a transaction without rolling back the entire transaction.

**Transaction Control**

In SQL Server, transaction control commands are used to manage transactions, ensuring data integrity and consistency. Here are the common commands:

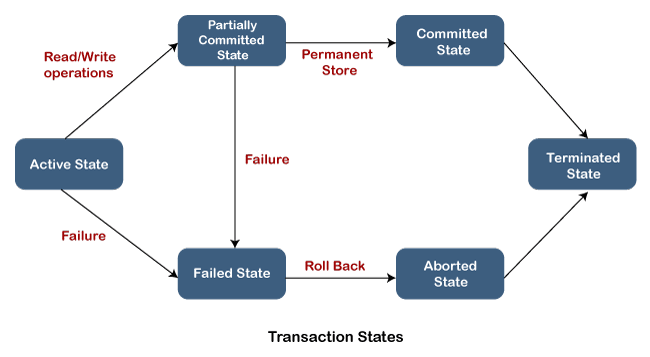
* **BEGIN TRANSACTION**: Starts a new transaction.
* **COMMIT TRANSACTION**: Saves the changes made during the current transaction to the database.
* **ROLLBACK TRANSACTION**: Undoes all changes made during the current transaction and restores the database to its previous state.
* **SAVE TRANSACTION**: Sets a savepoint within the current transaction, allowing partial rollback to that point if needed.
* **SET TRANSACTION**: Sets characteristics for the transaction, such as isolation level and whether it's read-only or read/write.

Note: for RELEASE SAVEPOINT, unlike other databases like Oracle, SQL Server doesn't have a specific RELEASE SAVEPOINT command. Instead, you can just issue a COMMIT or ROLLBACK to remove the savepoint. Once a transaction is committed or rolled back, all savepoints within that transaction are automatically released.

So, in SQL Server, you set a savepoint with SAVE TRANSACTION, rollback to it with ROLLBACK TRANSACTION, and release it implicitly when the transaction is committed or rolled back.

### Transaction State

It indicates how transactions go during their lifetime. **It describes the current state of the transaction as well as how the transaction will be processed in the future.** These states define the rules that determine whether a transaction commits or aborts.



**Active State:** The transaction is in an active state while the transaction's instructions are being executed. It will change to the **"partially committed state"** if all "read and write" operations are completed without errors. If any instruction fails, it changes to the "failed state."

**Partially Committed:** When all the read and write operations are completed, the change is made to the main memory or local buffer. The state would go to **"committed state"** if the changes are made permanent on the database. Otherwise, it goes to the "failed state".

**Failed State:** A transaction goes to the failed state when any transaction's instruction fails or a permanent modification on the database fails.

**Aborted State:** The transaction moves from a **"failed state"** to an **"aborted state"** when any kind of failure occurs. The changes are removed or rolled back because these changes are only made to the local buffer or main memory in previous states.

**Committed State:** A transaction is complete and goes into this state when the changes are made permanent on the database and terminated in the **"terminated state".**

**Terminated State:** If there is no rollback and the transaction is in the **"committed state,"** the system is consistent and ready for a new transaction while the old one is terminated.

Example:

CREATE TABLE Employee (

EmployeeID INT PRIMARY KEY,

Name NVARCHAR(100),

Department NVARCHAR(100),

Salary DECIMAL(10, 2)

);

INSERT INTO Employee (EmployeeID, Name, Department, Salary)

VALUES (1, 'John Doe', 'IT', 50000.00),

(2, 'Jane Smith', 'HR', 45000.00),

(3, 'Michael Johnson', 'Finance', 60000.00);

**Note1:** in most database management systems (including SQL Server), individual SQL statements such as CREATE TABLE, INSERT INTO, DELETE, and UPDATE are typically automatically committed after execution if they are not explicitly wrapped within a transaction block (i.e., BEGIN TRANSACTION and COMMIT TRANSACTION or ROLLBACK TRANSACTION).

This behavior is often referred to as "autocommit mode". In autocommit mode, each individual SQL statement is treated as a separate transaction that is automatically committed after it completes successfully.

Here:

-- CREATE TABLE statement outside of a transaction block

CREATE TABLE Employee (

EmployeeID INT PRIMARY KEY,

Name NVARCHAR(100),

Department NVARCHAR(100),

Salary DECIMAL(10, 2)

);

The table is created and the transaction is automatically committed. Or:

-- INSERT INTO statement outside of a transaction block

INSERT INTO Employee (EmployeeID, Name, Department, Salary)

VALUES (1, 'John Doe', 'IT', 50000.00);

The row is inserted into the table and the transaction is automatically committed. Or:

-- DELETE statement outside of a transaction block

DELETE FROM Employee WHERE EmployeeID = 1;

The row is deleted from the table and the transaction is automatically committed. Or:

-- UPDATE statement outside of a transaction block

UPDATE Employee SET Salary = 55000.00 WHERE EmployeeID = 1;

The row is updated and the transaction is automatically committed.

In each of above cases, the operation is committed immediately after the statement is executed successfully, without the need for explicit transaction management. However, if an error occurs during the execution of any of these statements, the transaction will be automatically rolled back, ensuring data consistency.

**Note2:** however, remember that, we can include the CREATE TABLE command within a transaction body in SQL Server. When we execute a CREATE TABLE statement inside a transaction, it will be part of that transaction's scope until the transaction is either committed or rolled back.

BEGIN TRANSACTION;

-- Create a sample table

CREATE TABLE Employee (

EmployeeID INT PRIMARY KEY,

Name NVARCHAR(100),

Department NVARCHAR(100),

Salary DECIMAL(10, 2)

);

-- Insert some sample data

INSERT INTO Employee (EmployeeID, Name, Department, Salary)

VALUES (1, 'John Doe', 'IT', 50000.00),

(2, 'Jane Smith', 'HR', 45000.00),

(3, 'Michael Johnson', 'Finance', 60000.00);

-- Check the data

SELECT \* FROM Employee;

-- Commit the transaction

COMMIT TRANSACTION;

And, in between we can have save-points:

BEGIN TRANSACTION;

--SAVE TRANSACTION s1;

CREATE TABLE Employee (

EmployeeID INT PRIMARY KEY,

Name NVARCHAR(100),

Department NVARCHAR(100),

Salary DECIMAL(10, 2)

);

--ROLLBACK TRANSACTION s1;

SAVE TRANSACTION s2;

INSERT INTO Employee (EmployeeID, Name, Department, Salary)

VALUES (1, 'John Doe', 'IT', 50000.00),

(2, 'Jane Smith', 'HR', 45000.00),

(3, 'Michael Johnson', 'Finance', 60000.00);

SELECT \* FROM Employee;

ROLLBACK TRANSACTION s2;

SELECT \* FROM Employee;

-- till not commited, we can rollback as per the savepoints

-- when commited, no rollback possible

COMMIT TRANSACTION;

**Note3:** we know that transactions are typically atomic, meaning that either all operations within the transaction succeed or none of them do. However, in the below example, even though the second INSERT statement violates the primary key constraint (assuming EmployeeID is meant to be unique), the table creation and the first INSERT statement are not rolled back automatically.

BEGIN TRANSACTION;

CREATE TABLE Employee (

EmployeeID INT PRIMARY KEY,

Name NVARCHAR(100),

Department NVARCHAR(100),

Salary DECIMAL(10, 2) );

INSERT INTO Employee (EmployeeID, Name, Department, Salary)

VALUES (1, 'John Doe', 'IT', 50000.00);

INSERT INTO Employee (EmployeeID, Name, Department, Salary)

VALUES (1, 'Jane Smith', 'HR', 45000.00);

COMMIT TRANSACTION;

The reason for this behavior is that the CREATE TABLE statement and the first INSERT statement are separate operations from the second INSERT statement within the same transaction. The failure of one operation within a transaction doesn't automatically roll back all preceding operations.

To achieve the desired behavior where all operations within the transaction are rolled back if any one of them fails, you would need to structure your transaction differently. You could use error handling to check for errors and roll back the transaction accordingly:

BEGIN TRANSACTION;

BEGIN TRY

-- Create a sample table

CREATE TABLE Employee (

EmployeeID INT PRIMARY KEY,

Name NVARCHAR(100),

Department NVARCHAR(100),

Salary DECIMAL(10, 2)

);

-- Insert some sample data

INSERT INTO Employee (EmployeeID, Name, Department, Salary)

VALUES (1, 'John Doe', 'IT', 50000.00),

(1, 'Jane Smith', 'HR', 45000.00); -- This will cause a primary key violation

-- Commit the transaction if all operations succeed

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- Rollback the transaction if an error occurs

ROLLBACK TRANSACTION;

PRINT 'An error occurred. Transaction rolled back.';

END CATCH;

Be mindful that in SQL Server, if the CREATE TABLE statement didn’t automatically rolled back like data modification statements (e.g., INSERT, UPDATE, DELETE), it is simply because DDL (Data Definition Language) statements like CREATE TABLE are not transactional in SQL Server by default.

However, you can prevent the creation of the table from being committed if an error occurs by explicitly checking for errors within the TRY...CATCH block and conditionally rolling back the transaction only if a DML (Data Manipulation Language) statement fails:

BEGIN TRANSACTION;

BEGIN TRY

-- Create a sample table

CREATE TABLE Employee (

EmployeeID INT PRIMARY KEY,

Name NVARCHAR(100),

Department NVARCHAR(100),

Salary DECIMAL(10, 2)

);

-- Check if the table was created successfully

IF OBJECT\_ID('Employee', 'U') IS NOT NULL

BEGIN

-- Insert some sample data

INSERT INTO Employee (EmployeeID, Name, Department, Salary)

VALUES (1, 'John Doe', 'IT', 50000.00),

(1, 'Jane Smith', 'HR', 45000.00); -- This will cause a primary key violation

END

-- Commit the transaction if all operations succeed

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- Rollback the transaction if an error occurs

ROLLBACK TRANSACTION;

PRINT 'An error occurred. Transaction rolled back.';

END CATCH;

In this modified example, before attempting to insert sample data into the Employee table, we check if the table was successfully created using the OBJECT\_ID function. If the table creation was successful, only then do we proceed with the data insertion. If an error occurs during either the table creation or the data insertion, the entire transaction is rolled back. This ensures that both DDL and DML operations are treated atomically within the transaction block.

OBJECT\_ID('Employee', 'U') is a function call in SQL Server that returns the object identification number (object\_id) of the specified database object. In above case, it checks whether a table named 'Employee' exists in the current database. The second argument 'U' specifies that it's looking for a user-defined table.

When OBJECT\_ID('Employee', 'U') is called, it returns NULL if the table 'Employee' doesn't exist in the database or if it's not a user-defined table. If the table exists, it returns the object\_id of the table.

The expression OBJECT\_ID('Employee', 'U') IS NOT NULL is used as a condition to check if the 'Employee' table exists in the database. If the condition evaluates to true, it means the table exists, and if it evaluates to false, it means the table doesn't exist.

**@@ERROR**

In SQL Server, @@ERROR is a system function that returns the error code produced by the last executed Transact-SQL statement. It is a global variable that holds the error number generated by the most recently executed SQL statement within the scope of the batch, stored procedure, trigger, or user-defined function.

Here's how @@ERROR works:

* After executing a SQL statement, you can check the value of @@ERROR to determine if an error occurred during the execution of that statement.
* If the value of @@ERROR is 0, it indicates that the last executed statement was successful, and no error occurred.
* If the value of @@ERROR is non-zero, it represents the error number generated by the SQL statement. You can then use this error number to perform error handling or logging as needed.

It's important to note a couple of things about @@ERROR:

1. Scope: The value of @@ERROR is scoped to the current session, meaning it retains its value until another SQL statement is executed, or until the session ends.
2. Resetting: The value of @@ERROR is reset after each SQL statement execution. Therefore, if you want to capture the error code for a specific statement, you should check the value of @@ERROR immediately after executing that statement.

DECLARE @ErrorCode INT;

-- Attempt to insert a record into a table

INSERT INTO TableName (Column1, Column2)

VALUES (Value1, Value2);

-- Check if an error occurred

SET @ErrorCode = @@ERROR;

-- If an error occurred, handle it

IF @ErrorCode <> 0

BEGIN

PRINT 'An error occurred. Error code: ' + CAST(@ErrorCode AS NVARCHAR(10));

-- Additional error handling logic can be added here

END

ELSE

BEGIN

PRINT 'Record inserted successfully.';

END

Above, @@ERROR is used to capture the error code generated by the INSERT statement. If an error occurs during the execution of the INSERT statement, the error code is stored in the @ErrorCode variable, and appropriate error handling logic can be executed based on the error code.

Example, demonstrating the use of transactions in SQL Server, including both commit and rollback scenarios. Suppose we have a table called Employee with columns EmployeeID, Name, and Salary. We want to update the salary of an employee and insert a new record into the table, ensuring that both operations succeed together.

-- Begin the transaction

BEGIN TRANSACTION;

-- Update the salary of an employee

UPDATE Employee SET Salary = 60000 WHERE EmployeeID = 123;

-- Insert a new employee record

INSERT INTO Employee (EmployeeID, Name, Salary) VALUES (456, 'John Doe', 55000);

-- Check if any errors occurred during the transaction

IF @@ERROR <> 0

BEGIN

-- If an error occurred, rollback the transaction

ROLLBACK TRANSACTION;

PRINT 'Transaction rolled back due to error.';

END

ELSE

BEGIN

-- If no errors occurred, commit the transaction

COMMIT TRANSACTION;

PRINT 'Transaction committed successfully.';

END

In this example:

* We begin the transaction using BEGIN TRANSACTION.
* We then update the salary of an employee with EmployeeID 123 and insert a new employee record with EmployeeID 456.
* We check if any errors occurred during the transaction using @@ERROR. If an error occurred (e.g., constraint violation, deadlock), we rollback the transaction using ROLLBACK TRANSACTION. Otherwise, if no errors occurred, we commit the transaction using COMMIT TRANSACTION.

Now, there are two scenarios:

Scenario 1: Successful Commit

Suppose both the update and insert operations complete successfully. When we run the script, it will print:

Transaction committed successfully.

In this case, the changes made by the transaction (salary update and new record insertion) are permanent and have been successfully committed to the database.

Scenario 2: Error and Rollback

Suppose an error occurs during the transaction, such as a constraint violation or deadlock. In this case, the @@ERROR check will detect the error, and the transaction will be rolled back. When we run the script, it will print:

Transaction rolled back due to error.

In this case, the changes made by the transaction (salary update and new record insertion) are rolled back, and the database is restored to its previous state before the transaction began.

And if we want to capture the error code:

BEGIN TRANSACTION;

-- Insert a new employee record

DECLARE @ErrorCode INT;

INSERT INTO Employee (EmployeeID, Name, Salary) VALUES (1, 'John Doe', 55000);

-- Capture the error code

SET @ErrorCode = @@ERROR;

-- Check if any errors occurred during the transaction

IF @ErrorCode <> 0

BEGIN

PRINT 'An error occurred. Error code: ' + CAST(@ErrorCode AS NVARCHAR(10));

-- If an error occurred, rollback the transaction

ROLLBACK TRANSACTION;

PRINT 'Transaction rolled back due to error.';

END

ELSE

BEGIN

-- If no errors occurred, commit the transaction

COMMIT TRANSACTION;

PRINT 'Transaction committed successfully.';

END

**Classification of Transactions**

Based on Nature:

* Read-Only Transactions: Transactions that only read data from the database but do not modify it. These transactions do not require the database to maintain locks for concurrency control.
* Read-Write Transactions: Transactions that read data from the database and also modify it. These transactions may require locks for concurrency control to ensure data consistency.

Based on Duration:

* Short Transactions: Transactions that complete quickly and involve a minimal number of database operations. These transactions typically don't hold resources for an extended period.
* Long Transactions: Transactions that take a considerable amount of time to complete and may involve complex operations or user interactions. Long transactions can potentially impact concurrency and resource usage in the database.

Based on Isolation Level:

Isolation levels determine how transactions interact with each other and with the data that other transactions are accessing or modifying. There are different isolation levels, each offering a different trade-off between concurrency and data consistency.

* Read Uncommitted: In this isolation level, transactions can read data that has been modified by other transactions but not yet committed. It offers the highest level of concurrency but may result in dirty reads (reading uncommitted data).

--in my own user:

--GRANT SELECT, INSERT, UPDATE, DELETE ON Employee TO aaa;

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

-- Transaction 1

BEGIN TRANSACTION;

SELECT \* FROM Employee; -- Reading uncommitted data

COMMIT TRANSACTION;

--through user aaa for example

-- Transaction 2

BEGIN TRANSACTION;

UPDATE Employee SET Salary = 55000.00 WHERE EmployeeID = 1; -- Data being modified

--at this moment, even if not committed, my own user sees the changes

COMMIT TRANSACTION;

* Read Committed: here, transactions can only read data that has been committed by other transactions. They do not allow dirty reads but still allows non-repeatable reads and phantom reads.

--in my own user:

--GRANT SELECT, INSERT, UPDATE, DELETE ON Employee TO aaa;

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Transaction 1

BEGIN TRANSACTION;

SELECT \* FROM Employee; -- Reading committed data

COMMIT TRANSACTION;

--through user aaa for example

-- Transaction 2

BEGIN TRANSACTION;

UPDATE Employee SET Salary = 55000.00 WHERE EmployeeID = 1; -- Data being modified

--at this moment, if not committed, my own user won’t see the changes. Hanging!

COMMIT TRANSACTION;