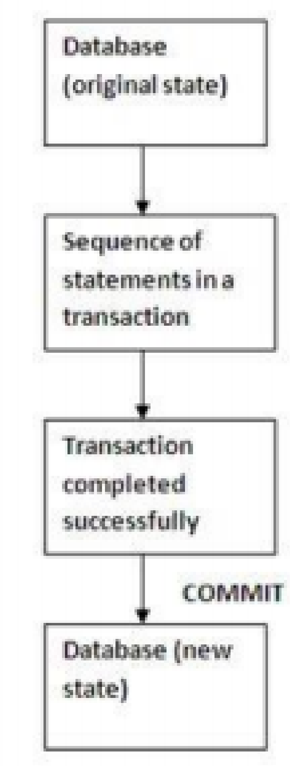
Transaction Management in SQL

A transaction is a unit of work performed against the database. It is a set of work (T-SQL statements) that are executed together such as a single unit in a specific logical order as a single unit.


Database (original state)
Sequence of statements in a transaction
Transaction completed successfully
COMMIT
Database (new state)

If statements are executed successfully then the transaction is complete and then it is committed that saves the data in the database permanently. If any single statement fails then the entire transaction will fail and the complete transaction will be cancelled or rolled back. When a transaction starts, it locks all the table data that is used in the transaction. Hence during the transaction life cycle no one can modify this table data used by the transaction such that the integrity of the data for the transaction is maintained.

A transaction is used when more than one table or view related to each other at a time are affected. The main goal of a transaction is for either all operations will be done or nothing will be done. We can compare a transaction with a digital circuit that works on 0 and 1. Here:

· 1 indicates completeness of all tasks (T-SQL statements) · 0 indicates no single tasks performed (T-SQL statements)

Example

A transaction is mainly used in banking or the transaction sector.

Let us see an example of a bank that has two customers, Cust_A and Cust_B. In case Cust_A wants to transfer some money to Cust_B, then there are the following 3 possibilities:

1. Debiting from the Cust_A account is performed successfully and crediting in the Cust_B account is performed successfully.
2. Neither debiting from the Cust_A account is performed nor crediting in the Cust_B account is performed.
3. Debiting from the Cust_A account is performed successfully, but crediting in the Cust_B account is not performed.

The first condition indicates a successful transaction and the second condition is not so critical. We are not required to do any retransmission, but the third condition will create a problem if, due to a technical problem, the first operation is successful but the second one fails. The result here would be that the Cust_A account will be debited, but the Cust_B account will not be credited. This means that we will lose the information.

For overcoming all these problems we can use transaction management. A transaction ensures that either a debit or a credit will be be done or nothing will be done.

Now we will explain what “Transaction Management “ is and how it works.

A transaction mainly consists of 4 properties that are also known as ACID

rules.

Atomicity: Atomic means that all the work in the transaction is treated as a single unit. Either it is performed completely or none of it is and at the point of failure the previous operations are rolled back to their former state.

Consistency: Transactions ensure that the database properly changes states upon a successfully committed transaction. In other words, if a transaction completes successfully then the database should be in a new state that will reflect changes else the transaction remains in the same state as at an initial point.

Isolation: It ensures that transactions operate independently and are transparent to each other. In other words, if more than one transections are running then they do not effect each other.
Durability: It ensures that the effect of committed transactions will save in the database permanently and should persist no matter what happens (like in a power failure).

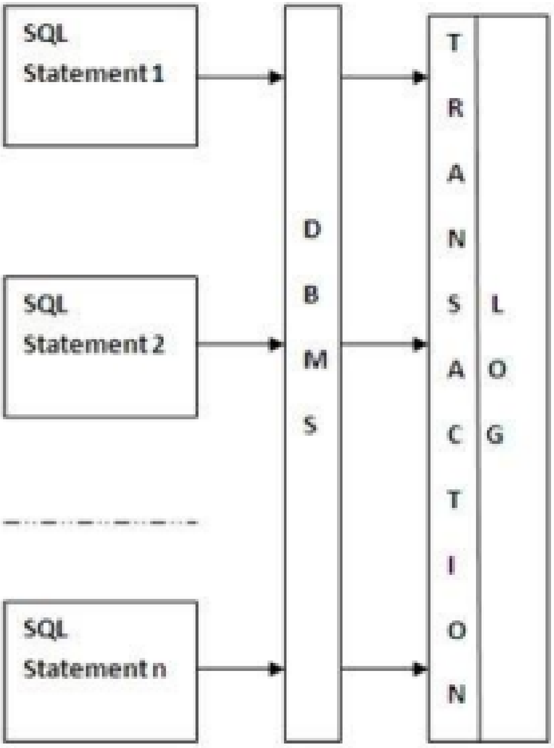
Types of Transactions

In SQL, transactions are of the following two types:

1. Implicit Transections
2. Explicit Transections

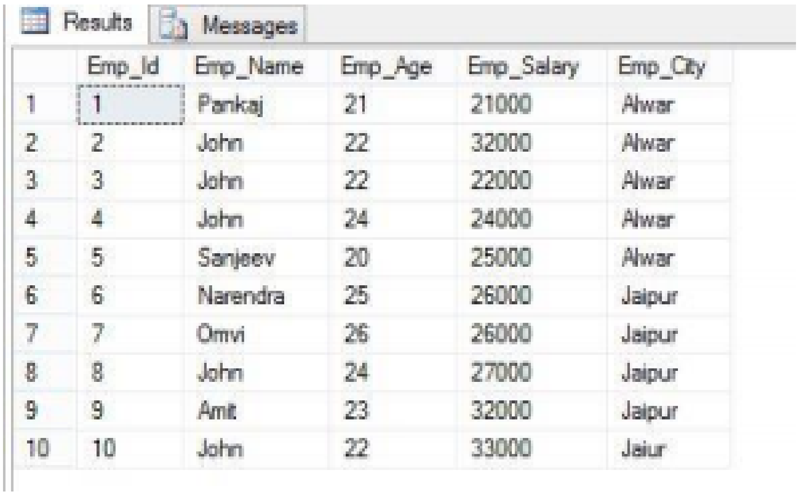
Implicit Transactions

Implicit transactions in the SQL language are performed by a DML query (insert, update and delete) and DDL query (alter, drop, truncate and create) statements. All these queries are handled by Implicit Transactions.



When any DDL or DML query is performed then the system stores the information of all the operations in the log file. If any error occurs then the SQL Server will rollback the complete statement.

Example



	Emp_Id	Emp_Name	Emp_Age	Emp_Salary	Emp_City
1	1	Pankaj	21	21000	Alwar
2	2	John	22	32000	Alwar
3	3	John	22	22000	Alwar
4	4	John	24	24000	Alwar
5	5	Sanjeev	20	25000	Alwar
6	6	Narendra	25	26000	Jaipur
7	7	Omvi	26	26000	Jaipur
8	8	John	24	27000	Jaipur
9	9	Amit	23	32000	Jaipur
10	10	John	22	33000	Jaiur

In the preceding table we have the following 5 columns:

Emp_Id datatype is Int, Emo_Name nvarchar(50) , Emp_Age int, Emp_Salary int, Emp_City int.

Now we will try to insert some values:

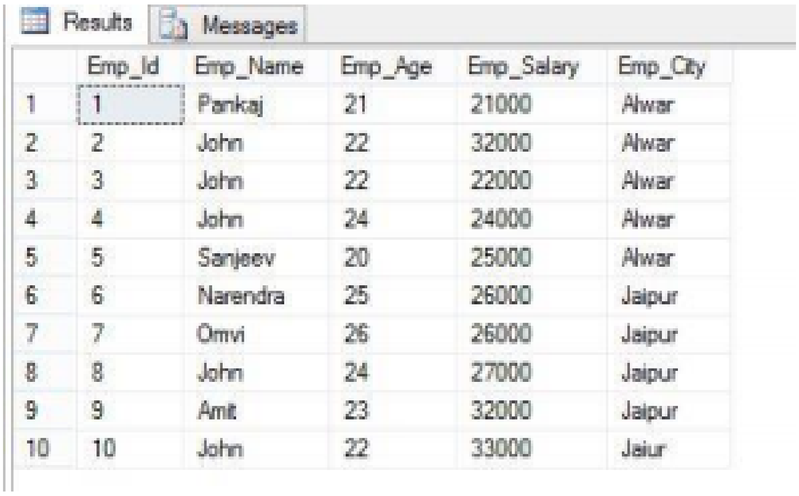
```
1. insert into Employee_Detail values(11,'Namo1',32,25000,'Delhi')
2. insert into Employee_Detail values(12,'Namo2',32,25000,'Delhi')
3. insert into Employee_Detail values(13,'Namo3','32,25000','Delhi') /* Here Error Will Occur */
4.
5.
6. insert into Employee_Detail values(14,'Namo4',32,25000,'Delhi')
7. insert into Employee_Detail values(15,'Namo5',32,25000,'Delhi')
```

Output

Msg 213, Level 16, State 1, Line 3 Column name or number of supplied values does not match the table definition. Now check the data of the table:

```
1. select * from Employee_Detail
```

Output



	Emp_Id	Emp_Name	Emp_Age	Emp_Salary	Emp_City
1	1	Pankaj	21	21000	Alwar
2	2	John	22	32000	Alwar
3	3	John	22	22000	Alwar
4	4	John	24	24000	Alwar
5	5	Sanjeev	20	25000	Alwar
6	6	Narendra	25	26000	Jaipur
7	7	Omvi	26	26000	Jaipur
8	8	John	24	27000	Jaipur
9	9	Amit	23	32000	Jaipur
10	10	John	22	33000	Jaiur

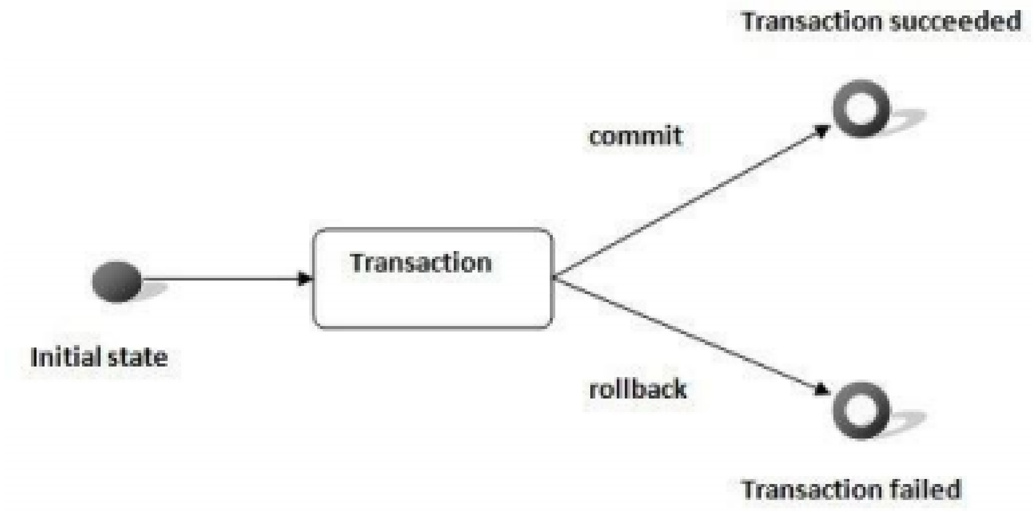
As we can see, in the preceding query the first two insertion queries worked correctly, but in the third statement an error occured. So a Transaction Rollback will be performed and the table will be restored to its initial state when the transaction was initiated.

It shows each DML and DDL query with the Transaction Control

Mechanism. **Explicit Transactions**

An explicit transaction is defined and controlled by the user on a DML query (insert, update or delete). A transaction is not applied on a SELECT command because is doesn't affect the data. A transaction is not used in creating tables or dropping them because these operations are automatically committed in the database.

Transaction Control
The following commands are used in the transaction control mechanism.



- **BEGIN:** To initiate a transaction.
- **COMMIT:** To save changes. After the commit command, the transaction can't rollback.
- **SAVEPOINT:** Provides points where the transaction can rollback to. · **ROLLBACK**: To rollback to a previous saved state.

Syntax of Transaction

Begin {Transaction|Tran }[Transaction_Name |@Trans_Name] Write Code

Here

End

Here

- **Begin:** Initiate transaction.
- **Transaction| Tran:** We can use any one out of both.
- **Transaction_Name:** Used for providing a name for a transaction. · **@Trans_Name:** This is the name of a user-defined variable containing a valid transaction name.
- **End:** Indicates the end of the transaction.

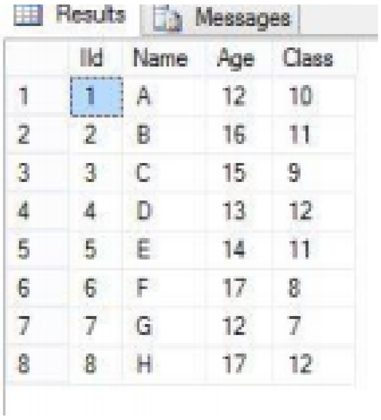
Now we will see some examples of Transaction Control Mechanisms. First, create a table:

```
Create Table Student
(
IId int Not Null primary key,
Name varchar(MAX) Not NUll,
Age Int Not Null,
Class int not Null
)
```

Now insert some values into the table:

```
Insert Into Student
Select 1,'A',12,10 Union All
Select 2,'B',16,11 Union All
Select 3,'C',15,9 Union All
Select 4,'D',13,12 Union All
Select 5,'E',14,11 Union All
Select 6,'F',17,8 Union All
Select 7,'G',12,7 Union All
Select 8,'H',17,12
```

Now the table will look like the following:



	IId	Name	Age	Class
1	1	A	12	10
2	2	B	16	11
3	3	C	15	9
4	4	D	13	12
5	5	E	14	11
6	6	F	17	8
7	7	G	12	7
8	8	H	17	12

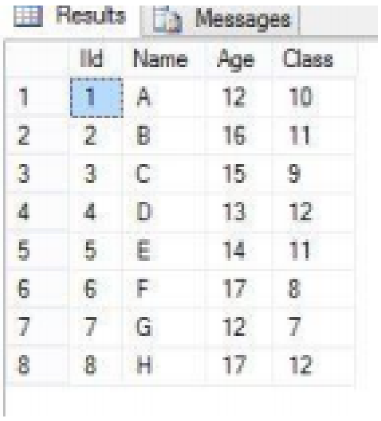
We will use the preceding table in the following example.

Example 1

```
Begin Transaction My_Trans
Delete from Student Where IId=3
Begin Rollback Transaction My_Trans
End
```

```
Select * From Student
```

Output



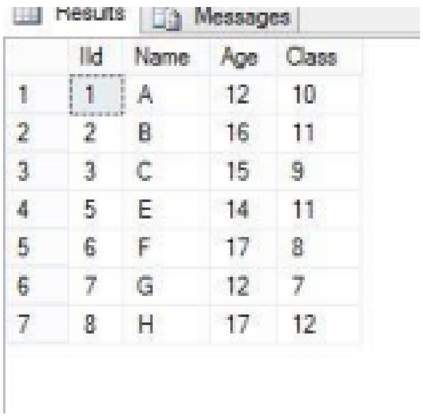
	IId	Name	Age	Class
1	1	A	12	10
2	2	B	16	11
3	3	C	15	9
4	4	D	13	12
5	5	E	14	11
6	6	F	17	8
7	7	G	12	7
8	8	H	17	12

In the preceding example we deleted a row from the table and then performed a rollback. Now we can see that there is no change in the table because we performed a rollback that transfers the table in the previous stable (saved) state.

Example 2

```
Begin Transaction My_Trans
Delete from Student Where IId=4
Commit Transaction My_Trans

Begin Transaction My_Trans
Rollback Transaction My_Trans

SELECT * FROM Student
```

Output



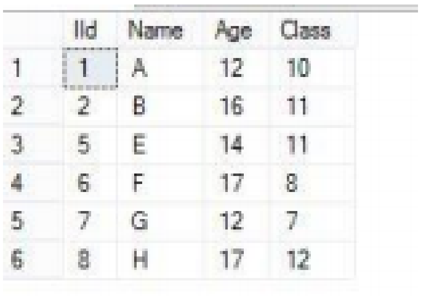
	IId	Name	Age	Class
1	1	A	12	10
2	2	B	16	11
3	3	C	15	9
4	5	E	14	11
5	6	F	17	8
6	7	G	12	7
7	8	H	17	12

In this example we deleted a row from the table and then performed a “commit” operation. The result of this operation saves all the changes that are performed. Now if we do a “Rollback” then the transaction will not be returned to its starting state because we commit (save) all the changes.

Example 3

```
Begin Transaction My_Trans

Delete from Student Where IId=3
save Transaction My_Save1

Delete from Student Where IId=4
save Transaction My_Save2

Delete from Student Where IId=5
save Transaction My_Save3

Begin Transaction My_Trans
Rollback Transaction My_Save2

SELECT * FROM Student
```

Output



	IId	Name	Age	Class
1	1	A	12	10
2	2	B	16	11
3	5	E	14	11
4	6	F	17	8
5	7	G	12	7
6	8	H	17	12

In a transaction we can create some savepointsthat saves the current state of the database. Then we can rollback to any specific savepoint. In the preceding example we performed 3 delete operations and after each delete operation we created a save point and finally we did a rollback at the save point ”My_Save2”. So, the operations performed after “My_save2” will be discarded.

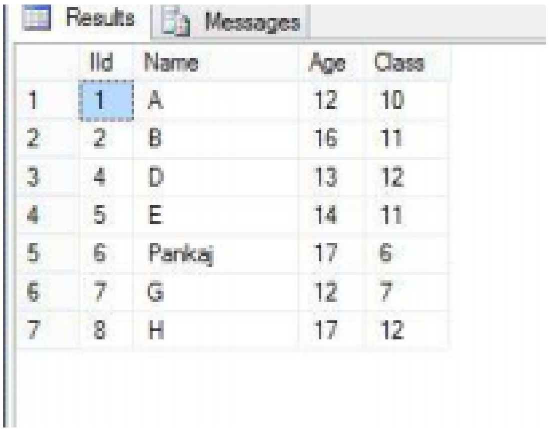
RELEASE SAVEPOINT:- This command is used to remove a SAVEPOINT that you have created. **Syntax:**

RELEASE SAVEPOINT SAVEPOINT_NAME
Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT. It is used to initiate a database transaction and used to specify characteristics of the transaction that follows.

Example 4

```
Begin Transaction Trans
Begin Try
Delete From Student Where Student.IId=3;
Update Student Set Student.Name='Pankaj' ,Student.Class=6 Where Student.IId=6
If @@TranCount>0
begin Commit Transaction Trans
End
End Try
Begin Catch
if @@TranCount>0
Print 'Error Is Occur in Transaction'
begin Rollback Transaction Trans
End
End Catch

Select * From Student
```

Output



	IId	Name	Age	Class
1	1	A	12	10
2	2	B	16	11
3	4	D	13	12
4	5	E	14	11
5	6	Pankaj	17	6
6	7	G	12	7
7	8	H	17	12

In this example we used a new concept of @@TRANCOUNT. @@TRANCOUNT returns the number of active transactions for the current connection. The starting value of @@TRANCOUNT is zero. When a new transaction begins, it increase its value by 1. When the commit statement decreases its value by 1 and the rollback statement decreases the value of @@TRANCOUNT to 0, the savepoint statement doesn't affect the value of @@TRANCOUNT.

Let us see an example:

```
Print @@Trancount
Begin Transaction
Print @@Trancount
Begin Transaction
Print @@Trancount
Commit Transaction
Print @@Trancount
Rollback Transaction
Print @@Trancount
```

Output

```
0
1
2
1
0
```

The following describes when to use @@TRANCOUNT:

1. In the case of exception handling and nested transactions. 2. The current transaction was called by some .NET code with its own transaction.
3. The current transaction was called from another Stored Procedure that had its own transaction.

The following is another example:

```
Begin Transaction Trans
Begin Try
Delete From Student Where Student.IId=3;
Update Student Set Student.Name=12121 ,Student.Class='12th'
 Where Student.IId=6 /* Error Will Occur Here */
If @@TranCount<0
begin Commit Transaction Trans
End
End Try
Begin Catch
if @@TranCount<0
Print 'Error Is Occur in Transaction'
begin Rollback Transaction Trans
End
End Catch
```

Output

```
(0 row(s) affected)Error Is Occur in Transaction
```

Example 5

```
Begin Transaction Trans
Begin Try
Update Student Set Student.Name='Pankaj' Where Student.Class=11
If @@ROWCOUNT=2
Print 'Number of Rows affected is 2, so rollback occurs'
begin RollBack Transaction Trans
End
End Try
Begin Catch
if @@ROWCOUNT=2
Print 'Error Is Occur in Transaction'
begin Rollback Transaction Trans
End
```

```
End Catch
```

Output

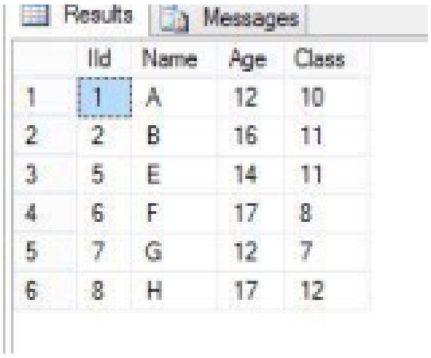
(2 row(s) affected)
Number of rows affected is 2, so rollback occurs.

@@ROWCOUNT is another important factor used in transactions. It returns the number of rows affected during the transaction. Using the value of @@ROWCOUNT we can use a proper action. In the preceding example the update query updates two rows, so the value of @@ROWCOUNT is 2 and we did a rollback.

Example 6

```
Create Procedure My_Proc__
AS
Begin Transaction
Delete From Student Where IId=3
Delete From Student Where IId='123' /* Error Occur Here */
Delete From Student Where IId=4

Commit Transaction
Go
EXEC My_Proc__
Go
Select * From Student
```

Output



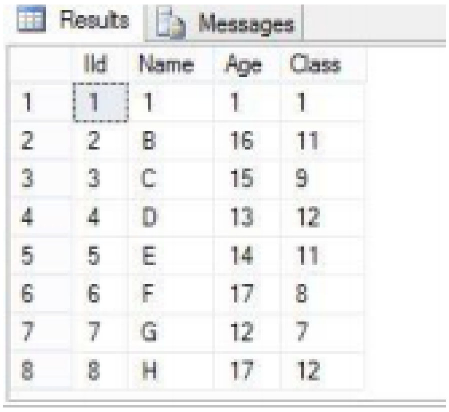
	IId	Name	Age	Class
1	1	A	12	10
2	2	B	16	11
3	5	E	14	11
4	6	F	17	8
5	7	G	12	7
6	8	H	17	12

This example shows that the transaction is not done correctly with the Stored Procedure. The problem with this Stored Procedure is that the transactions don't care if the statements run correctly or not. They only care is if SQL

Server fails in the middle. In the preceding example if an error occurs then the transaction is still going on. So, it is our responsibility to check for an error after each step in the Stored Procedure. Now we will see how to do this.

```
Create Procedure My_Proc_
AS
Begin Transaction My_Trans
Insert Into Student Values(9,'1',50,11)

If @@ERROR <>0
Begin
Rollback Transaction My_Trans
Return 4
End
Insert Into Student Values(9,'J',15,11) /* Error Occur Here */
If @@ERROR <>0
Begin
Rollback Transaction My_Trans
Return 4
End
Insert Into Student Values(11,'K',50,11)
If @@ERROR <>0
Begin
 Rollback Transaction My_Trans
Return 4
End
Commit Transaction My_Trans
Go
EXEC My_Proc_
Go
Select * From Student
```

Output



	IId	Name	Age	Class
1	1	1	1	1
2	2	B	16	11
3	3	C	15	9
4	4	D	13	12
5	5	E	14	11
6	6	F	17	8
7	7	G	12	7
8	8	H	17	12

In the preceding example the first insert query executes with no error, but in the second insert query an error will occur, violation of “**PRIMARY KEY constraint 'PK__Student__C4972BAC3D5E1FD2'. Cannot insert duplicate key in object 'dbo.Student'. The duplicate key value is (9).**” So the value of @@ERROR is not equal to 0 if the condition becomes true, so a rollback will be done. The @@ERROR system function returns 0 if the last Transact-SQL statement executed successfully; if the statement generated an error, @@ERROR returns the error number.

Example 7

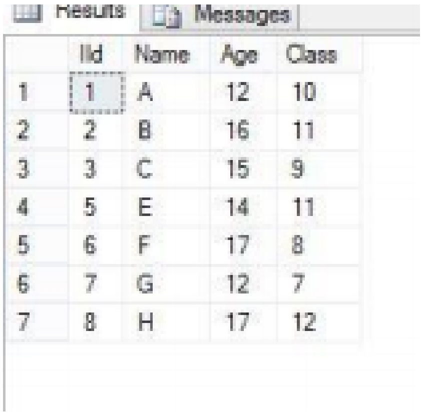
```
Declare @Trans_Name nvarchar(50)
Set @Trans_Name='My_Trans'
Begin Transaction @Trans_Name
Delete from Student Where IId=4
Commit Transaction @Trans_Name

Begin Transaction @Trans_Name
Rollback Transaction @Trans_Name

SELECT * FROM Student
```

In this example we defined the syntax of the transaction that shows that we can provide the name of a transaction using a user-defined variable.

@Trans_Name: This is the name of a user-defined variable containing a valid transaction name.

Here the user-defined variable @Trans_Name contains the name of the transaction.

Output



	IId	Name	Age	Class
1	1	A	12	10
2	2	B	16	11
3	3	C	15	9
4	5	E	14	11
5	6	F	17	8
6	7	G	12	7
7	8	H	17	12

The Transaction Log

Every SQL Server database has a transaction log that records all transactions and the database modifications made by each transaction.

The transaction log is a critical component of the database. If there is a system failure, you will need that log to bring your database back to a consistent state.

The transaction log supports the following operations:

- Individual transaction recovery.
- Recovery of all incomplete transactions when SQL Server is started.
- Rolling a restored database, file, filegroup, or page forward to the point of failure.
- Supporting transactional replication.
- Supporting high availability and disaster recovery solutions: Always On availability groups, database mirroring, and log shipping.

1. Individual transaction recovery

If an application issues a ROLLBACK statement, or if the Database Engine detects an error such as the loss of communication with a client, the log records are used to roll back the modifications made by an incomplete transaction.

2. Recovery of all incomplete transactions when SQL Server is started

If a server fails, the databases may be left in a state where some modifications were never written from the buffer cache to the data files, and there may be some modifications from incomplete transactions in the data files. When an instance of SQL Server is started, it runs a recovery of each database. Every modification recorded in the log that may not have been written to the data files is rolled forward. Every incomplete transaction found in the transaction log is then rolled back to make sure the integrity of the database is preserved.

3. Rolling a restored database, file, filegroup, or page forward to the point of failure

After a hardware loss or disk failure affecting the database files, you can restore the database to the point of failure. You first restore the last full database backup and the last differential database backup, and then restore the subsequent sequence of the transaction log backups to the point of failure.

As you restore each log backup, the Database Engine reapplies all the modifications recorded in the log to roll forward all the transactions. When the last log backup is restored, the Database Engine then uses the log information to roll back all transactions that were not complete at that point.

4. Supporting transactional replication

The Log Reader Agent monitors the transaction log of each database configured for transactional replication and copies the transactions marked for replication from the transaction log into the distribution database.

5. Supporting high availability and disaster recovery solutions

The standby-server solutions, Always On availability groups, database mirroring, and log shipping, rely heavily on the transaction log.
In an **Always On availability groups scenario**, every update to a database, the primary replica, is immediately reproduced in separate, full copies of the database, the secondary replicas. The primary replica sends each log record immediately to the secondary replicas, that applies the incoming log records to availability group databases, continually rolling it forward.

In a **log shipping scenario**, the primary server sends the active transaction log of the primary database to one or more destinations. Each secondary server restores the log to its local secondary database.

In a **database mirroring scenario**, every update to a database, the principal database, is immediately reproduced in a separate, full copy of the database, the mirror database. The principal server instance sends each log record immediately to the mirror server instance, which applies the incoming log records to the mirror database, continually rolling it forward.

Transaction log characteristics

Characteristics of the SQL Server Database Engine transaction log:

- The transaction log is implemented as a separate file or set of files in the database. The log cache is managed separately from the buffer cache for data pages, which results in simple, fast, and robust code within the SQL Server Database Engine.
- The format of log records and pages is not constrained to follow the format of data pages.
- The transaction log can be implemented in several files. The files can be defined to expand automatically by setting the FILEGROWTHvalue for the log. This reduces the potential of running out of space in the transaction log, while at the same time reducing administrative overhead..
- The mechanism to reuse the space within the log files is quick and has minimal effect on transaction throughput.
- Log truncation frees space in the log file for reuse by the transaction log.