TRANSACTION PROCESSING

Engr. Laraib Siddiqui

Transaction

A ***transaction*** is a *unit* of program execution that accesses and possibly updates various data items.

- A transaction must see a consistent database.
- During transaction execution the database may be inconsistent.
- When the transaction is committed, the database must be consistent.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties

Transactions should possess several properties, often called the **ACID** properties; they should be enforced by the concurrency control and recovery methods of the DBMS.

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** if the database is in a consistent state before executing the transaction, it will be in a consistent state after the *complete* execution of the transaction, assuming that *no interference with other transactions* occurs.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Why Atomicity?

Account(Ano, Name, Type, Balance)

A user want to

update Account set Balance = Balance –50 where Ano = 10001

update Account set Balance = Balance + 50 where Ano = 12300

System crashed in the middle

Possible outcome w/o recovery:

$50 transferred or lost

The operations must be done as a unit

Why Isolation?

Two users (programs) do this at the same time

- User 1: update Student set GPA = 3.7 where SID = 123
- User 2: update Student set Major = 'CS' where SID = 123

Sequence of events: for each user, read tuple, modify attribute, write tuple.

Possible outcomes w/o concurrency control: one change or both

Example of Fund Transfer

Transaction to transfer $50 from account *A* to account *B*:
1.read(A)
2.A:= A –50
3.write(A)
4.read(B)
5.B:= B + 50
6.write(B)

Consistency requirement –the sum of *A* and *B* is unchanged by the execution of the transaction.

Atomicity requirement —if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

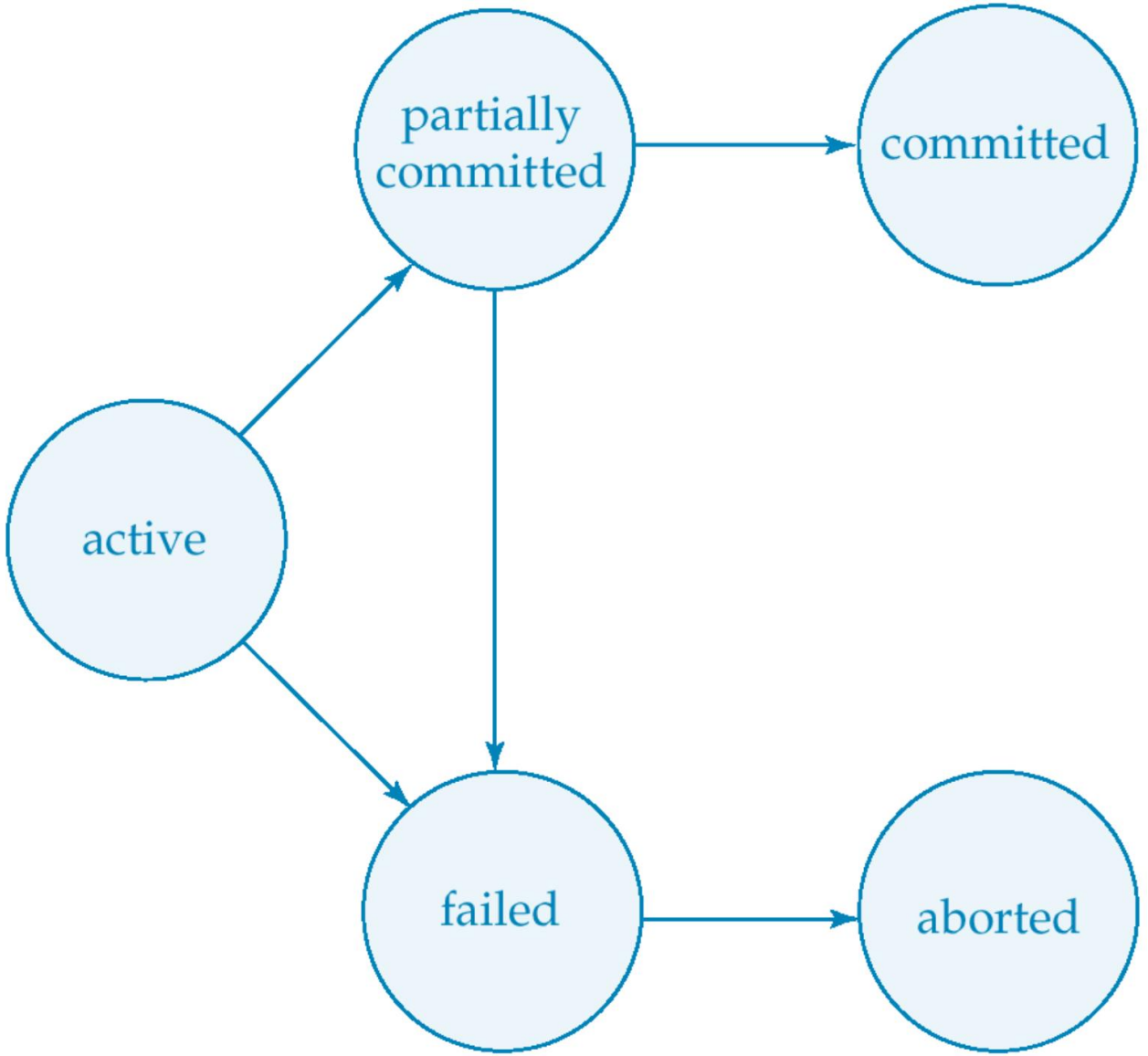
Example of Fund Transfer

Durability requirement —once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.

Isolation requirement —if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum A + B will be less than it should be).Can be ensured trivially by running transactions serially, that is one after the other. However, executing multiple transactions concurrently has significant benefits, as we will see.

Transaction State

- **Active,** the initial state; the transaction stays in this state while it is executing
- **Partially committed,** after the final statement has been executed.
- **Failed,** after the discovery that normal execution can no longer proceed.
- **Aborted,** after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction –only if no internal logical error
  - kill the transaction
- **Committed,** after *successful completion*.


partially committed
committed
active
failed
aborted

Concurrent Executions

Multiple transactions are allowed to run concurrently in the system. Advantages are:

- **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
- **reduced average response time** for transactions: short transactions need not wait behind long ones.
- *Concurrency control schemes*–mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedule: Modeling Concurrency

Schedule: a sequence of operations from a set of transactions, where operations from any one transaction are in their original order

Ri(X): read X by Ti

Wi(X): write X by Ti

$R_1(A), W_1(A), R_2(B), W_2(B), R_1(C), W_1(C)$

In a complete schedule, each transaction ends in commit or abort.

A schedule transforms database from an initial state to a final state

T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	

Schedule

Assume a consistent initial state

A representation of an execution of operations from a set of transactions

- Ignore
  - aborted transactions
  - Incomplete (not yet committed) transactions
- Operations in a schedule conflict if
  - They belong to different transactions
  - They access the same data item
  - At least one item is a write operation

Anomalies with Concurrency

Interleaving transactions may cause many kinds of consistency problems

Reading Uncommitted Data ("dirty reads"):

R1(A), W1(A), R2(A), W2(A), C2, R1(B), C1

Unrepeatable Reads:

R1(A), R2(A), W2(A), C2, R1(A), W1(A), C1

Overwriting Uncommitted Data (lost update):

R1(A), R2(A), W2(A), W1(A)

Anomalies with Concurrency

Incorrect Summary Problem

- Data items may be changed by one transaction while another transaction is in the process of calculating an aggregate value
- A correct "sum" may be obtained prior to any change, or immediately after any change

Serial Schedule

An acceptable schedule must transform database from a consistent state to another consistent state

Serial schedule: one transaction runs entirely before the next transaction starts.

T_1: R(X), W(X)

T_2: R(X), W(X)

$R_1(X)\ W_1(X)\ C_1 R_2(X)\ W_2(X)\ C_2$

$R_2(X)\ W_2(X)\ C_2 R_1(X)\ W_1(X)\ C_1$

} **Serial**

$R_1(X)\ R_2(X)\ W_2(X)\ W_1(X)\ C_1\ C_2$

} **Non-serial**

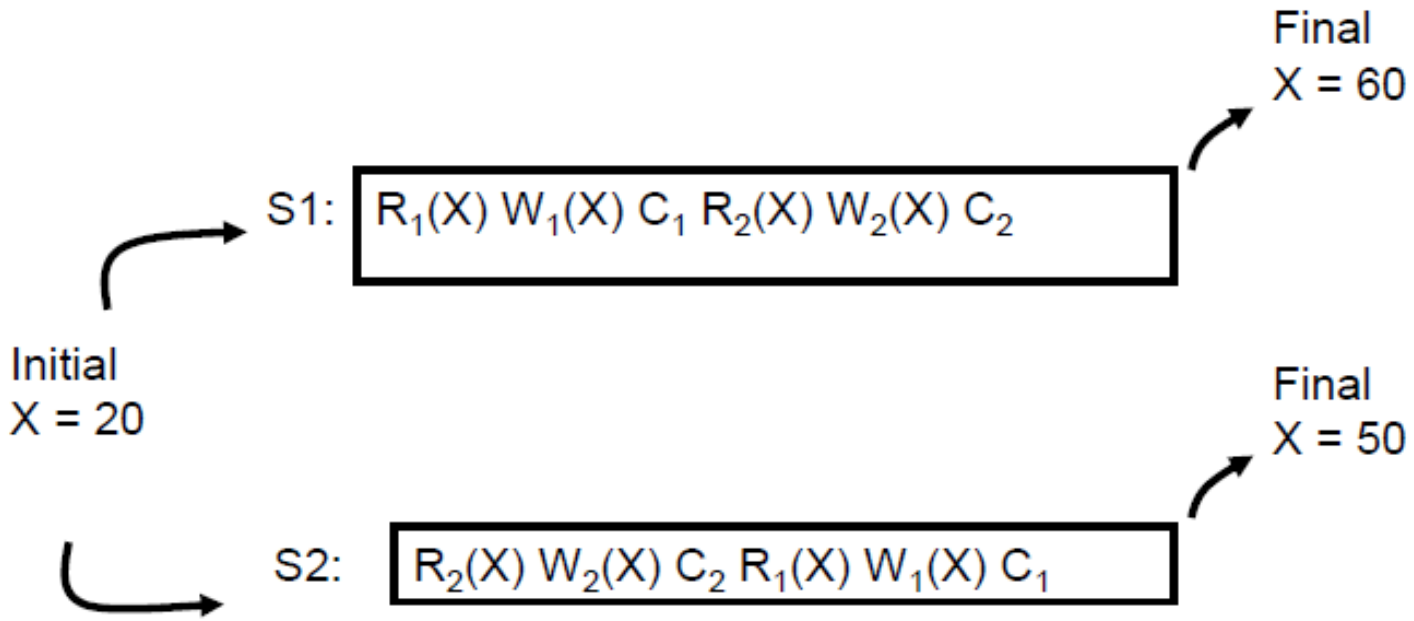
Serial schedules guarantee transaction isolation & consistency

Different serial schedules can have different final states

Serial Schedules

T1: R(X), X=X+10, W(X)

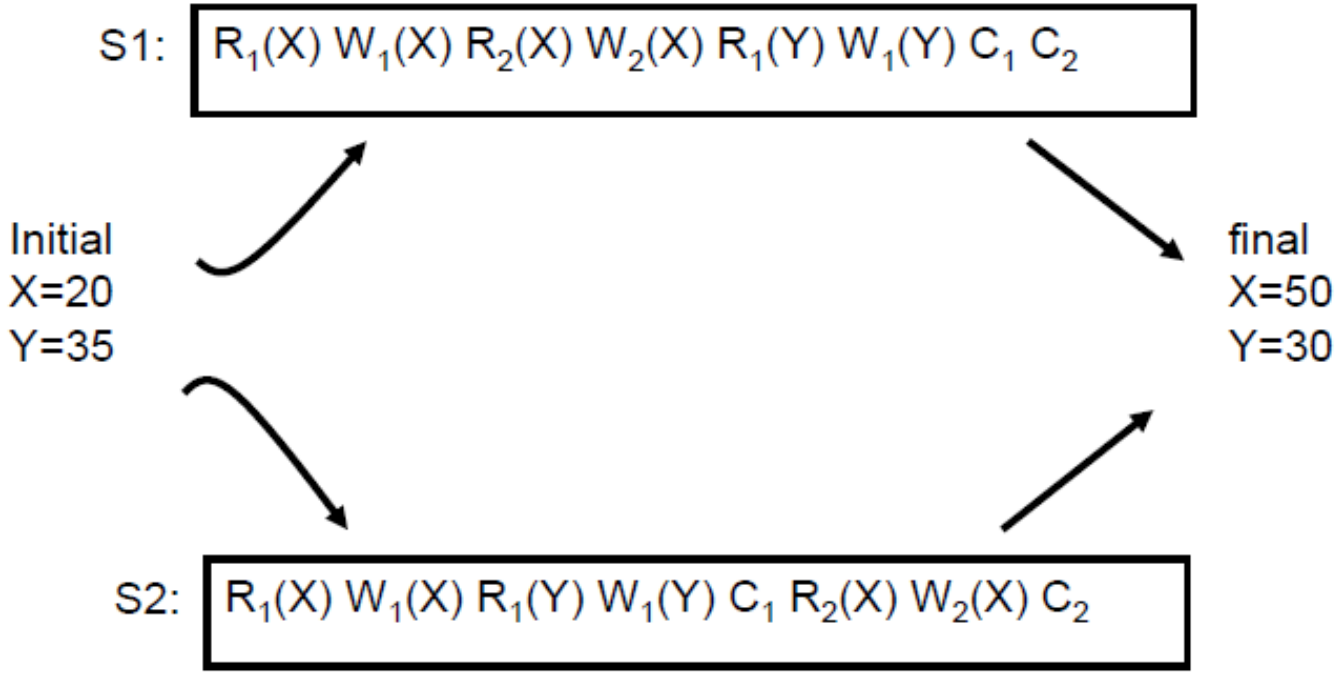
T2: R(X), X=X*2, W(X)



Non-Serial Schedule

T1: R(X), X=X*2, W(X), R(Y), Y=Y-5, W(Y)

T2: R(X), X=X+10, W(X)



Schedule

Let *T*1 transfer $50 from *A* to *B*, and *T*2 transfer 10% of the balance from *A* to *B*.

Serial schedule in which *T*1is followed by *T*2

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Serial schedule where T2is followed by T1

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Not a serial schedule, but it is equivalent

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Schedule

The following concurrent schedule does not preserve the value of (*A* + *B*).

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Serializability

Serializability ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order. It assumes that all accesses to the database are done using read and write operations. A schedule is called ``correct'' if we can find a serial schedule that is ``equivalent'' to it.

Different forms of schedule equivalence give rise to the notions of:

1. Conflict serializability
2. View serializability

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflict Serializable

A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

Conflicting operations: Two operations are said to be conflicting if all conditions satisfy:

- They belong to different transactions
- They operate on the same data item
- At Least one of them is a write operation

Conflict Equivalent

If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´*are conflict equivalent.

We say that a schedule *S* is conflict serializable if it is conflict equivalent to a serial schedule.

Schedule 1 can be transformed into Schedule 2, a serial schedule where T2 follows T1, by series of swaps of non-conflicting instructions.

Therefore Schedule 1 is conflict serializable.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Test for Conflict Serializability

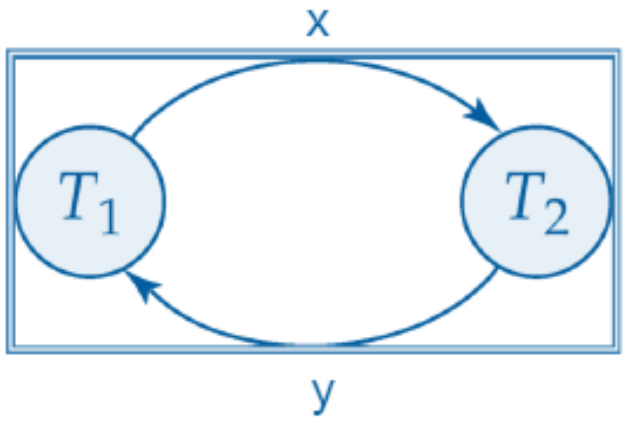
A schedule is conflict serializable if and only if its precedence graph is acyclic.

Consider some schedule of a set of transactions *T1*, *T2*, …, *Tn*

Precedence graph—a direct graph where the vertices are the transactions (names).

We draw an arc from *T1* to *T2* if the two transaction conflict.

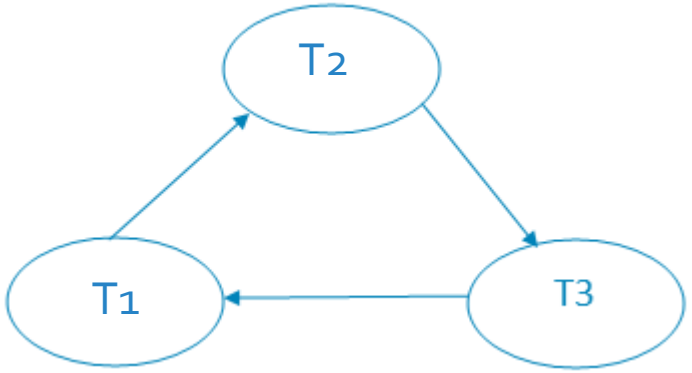
We may label the arc by the item that was accessed.



Test for Conflict Serializability

Time

T1	T2	T3
Read(A)		
	Read(B)	
$A := f_1(A)$		
		Read(C)
	$B := f_2(B)$	
	Write(B)	
		$C := f_3(C)$
		Write(C)
Write(A)		
		Read(B)
	Read(A)	
	$A := f_4(A)$	
Read(C)		
	Write(A)	
$C := f_5(C)$		
Write(C)		
		$B := f_6(B)$
		Write(B)

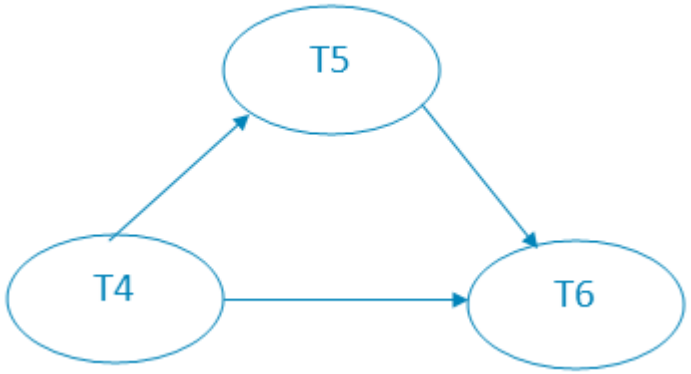


The precedence graph for schedule contains a cycle that's why Schedule is non-serializable.

Test for Conflict Serializability

T4	T5	T6
Read(A)		
A:= f1(A)		
Read(C)		
Write(A)		
A:= f2(C)		
	Read(B)	
Write(C)		
	Read(A)	
		Read(C)
	B:= f3(B)	
	Write(B)	
		C:= f4(C)
		Read(B)
		Write(C)
	A:=f5(A)	
	Write(A)	
		B:= f6(B)
		Write(B)

Time



The precedence graph for schedule contains no cycle that's why Schedule is serializable.

Check for indegree = 0

Equivalent serial schedule

T4 -> T5 -> T6

Practice

Construct a precedence graph for given schedule. Show whether the following schedules are conflict serializable. Also select the possible serial schedule for this graph.

T_1	T_2	T_3
	R(A)	
R(B)		
	W(A)	
	R(B)	
		R(A)
W(B)		
		W(A)
	W(B)	

View Serializability

A schedule will view serializable if it is view equivalent to a serial schedule.

Two schedules are said to be view equivalent if the order of initial read, final write and update operations is the same in both the schedules.

Every conflict serializable schedule is also view serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Recoverable Schedules

If a transaction *Tj* reads a data item previously written by a transaction *Ti* , then the commit operation of *Ti* appears before the commit operation of *Tj.*

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

For above schedule to be recoverable, *T7* would have to delay committing until after *T6* commits.

Cascadeless Schedules

Even if a schedule is recoverable, to recover correctly from the failure of a transaction *Ti*, we may have to roll back several transactions.

A single transaction failure leads to a series of transaction rollbacks, is called cascading rollback.

Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable). If *T*10 fails, *T*11 and *T*12 must also be rolled back.

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called *cascadeless* schedules.

Transaction in SQL Server

1. **Autocommit Transaction** mode is the default transaction for the SQL Server. In this mode, each T-SQL statement is evaluated as a transaction and they are committed or rolled back according to their results. The successful statements are committed and the failed statements are rolled back immediately
2. **Implicit transaction** mode enables to SQL Server to start an implicit transaction for every DML statement but we need to use the commit or rolled back commands explicitly at the end of the statements
3. **Explicit transaction** mode provides to define a transaction exactly with the starting and ending points of the transaction

Transaction in SQL Server Example

```
BEGIN TRANSACTION MyTransaction
BEGIN TRY
UPDATE Account SET Debit=100 WHERE Name=‘Ali'
UPDATE ContactInformation SET Mobile='1234567890' WHERE Name=‘Haider'
COMMIT TRANSACTION MyTransaction
PRINT 'TRANSACTION SUCCESS'
END TRY
BEGIN CATCH
ROLLBACK TRANSACTION MyTransaction
PRINT 'TRANSACTION FAILED'
END CATCH
```

Example

```
SET IMPLICIT_TRANSACTIONS ON

UPDATE   Person  SET   Lastname = 'Sawyer',   Firstname = 'Tom' WHERE  PersonID = 2 ;

DELETE   Person  WHERE PersonID = 1;

SELECT   @@TRANCOUNT AS OpenTransactions
```