Transaction

Chapter: 17

Topics to be Covered

- Concept of Transaction
- States of Transaction
- ACID Properties
- Problems associated with Concurrency
  - Dirty Read Problem
  - Lost Update Problem
  - Unrepeatable read problem
- **Schedule**

Basic of Transaction

- A transaction in a database is a **single unit of work** that consists of one or more SQL operations (like *INSERT, UPDATE, DELETE*) executed together.
- The purpose of a transaction is to ensure **data integrity**, **consistency**, and **reliability**, especially in multi-user and ***concurrent*** environments.

Basic of Transaction

- A Transaction is a logical unit of work.
- It is the set of operations(basically ***read*** and ***write***) to perform unit of work,(include small units of work)
- Transaction which successfully completes its execution is said to have been ***committed***
- otherwise the transaction is ***aborted*** or ***rollback***

Examples of Transaction

Transaction: Transfer 500 from one account to another account

```
UPDATE accounts SET balance = balance - 500
WHERE name = 'Alice';
UPDATE accounts SET balance = balance + 500
WHERE name = ‘Charlie’;
```

Problem : If the system crashes after deducting from Alice’s account but before adding to Charlie’s account, money is lost.

Examples of Transaction (Cont..)

Solution: Use transaction

```
BEGIN;
    UPDATE accounts SET balance = balance - 500 WHERE
    name = 'Alice';
    UPDATE accounts SET balance = balance + 500 WHERE
    name = 'Charlie';
COMMIT;
```

- If all operations are successful, the COMMIT command makes the changes permanent.
- If something goes wrong (e.g., network failure or error), the transaction can be rolled back:

  ROLLBACK;

Example of Transaction

T1	T2
R(A)	
A=A+100	
	R(A)
	A=A-50
	W(A)
W(A)	
Commit(T1)	
	Commit(T2)

Real Life Examples

- Banking System
- Online Booking System (Hotels, Tickets)
- E-commerce systems
- Student Registration systems

Problems Associated with Transaction

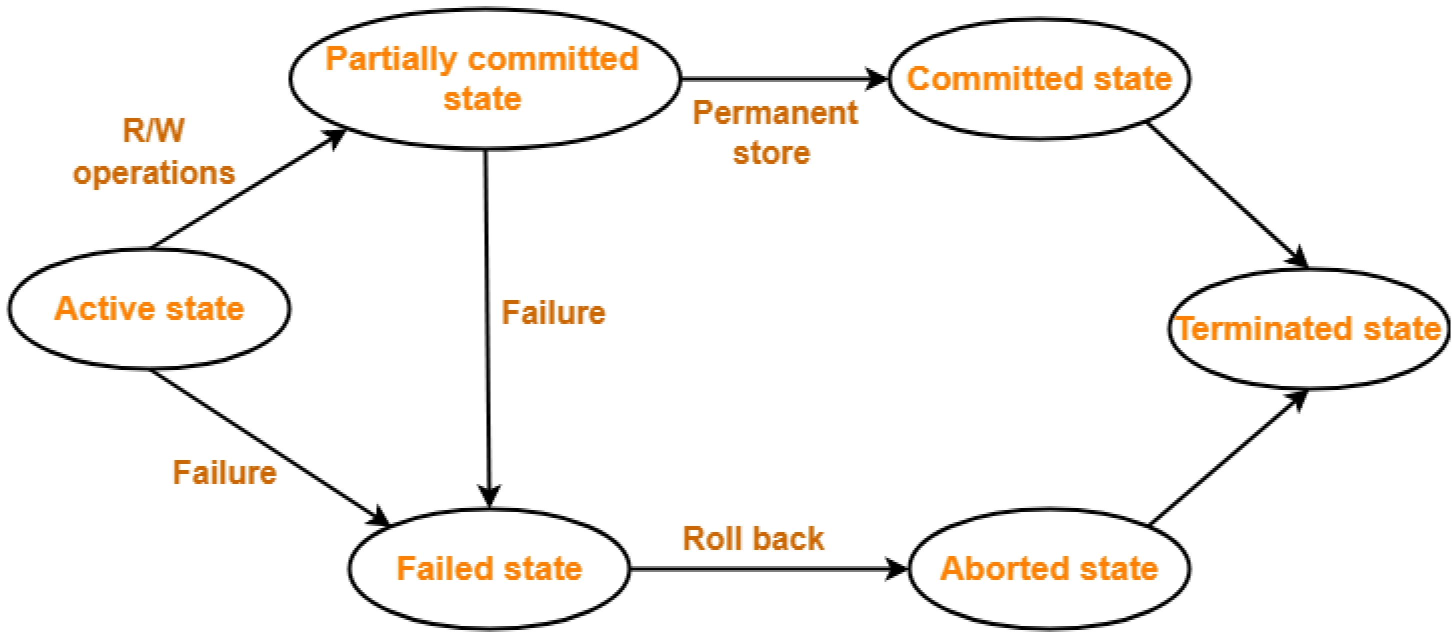
Without having Transaction a system may face the following problems :

- It may create an inconsistent results.
- It may create problems in concurrent execution.
- It may create an uncertainty to decide when to make changes permanent.

How to solve the above 3 problems ?: ACID properties

- **Atomicity: All or Nothing**
  - Atomicity ensures that all the operations in a transaction are **executed completely** or **none at all**.
- **Consistency: Valid state before and after**
  - A transaction must take the database from **one consistent state to another consistent state**, maintaining all **data integrity rules** (e.g., constraints, foreign keys).
- **Isolation: No Interference between transactions**
  - Each transaction must execute **independently of others**, even when multiple transactions are running concurrently.
- **Durability: Permanent results**
  - Once a transaction is **committed**, its effects are **permanent**, even if the system crashes immediately afterward.
  - RAID(Redundant Array of Independent Disks)

Transaction States



Transaction States in DBMS

Advantages of **Concurrency**

- **Parallel vs Concurrent**
- Decrease waiting time
- Decrease response time/ increase performance
- Resource utilization
- Increase efficiency

Problems with concurrency

- Dirty Read Problem/Uncommitted dependency/ Temporary update
- Lost Update Problem (Write-Write Conflict)
- Unrepeatable read problem
- Phantom read problem (Close to URP)
- Incorrect Summary Problem

Dirty Read Problem

Transaction T1	Transaction T2
R (A)	
W (A)	
	R (A) // Dirty Read
	W (A)
	Commit
Failure	

Dirty Read Problem

- Solution
  - Ignore these type of scenario
  - Don't read any value from local buffer
  - Read value directly from memory if needed
  - If read from local buffer before committing wait for previous transaction
  - Read value after committing
  - If read then ensure there is no possibility of failure

Lost Update Problem

Time	Tx	Ty
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

LOST UPDATE PROBLEM

Unrepeatable read problem

Transaction T1	Transaction T2
R (X)	
	R (X)
W (X)	
	R (X) // Unrepeated Read

Phantom Read Problem

T1	**T2**
READ(A)	
	READ(A)
DELETE(A)	
	READ(A) // *A is missing*

Schedule

- Order of transaction is called schedule.
- Two types: ***Serial schedule*** and ***Non serial or concurrent schedule***
- SS-> less throughput, no problem with isolation
- CS-> need to manage isolation

Example of Schedule

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Serial Schedule

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

Non Serial/Concurrent Schedule

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)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

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)	
commit	
	$B := B + temp$
	write(B)
	commit

Inconsistent State

Conflict Serializable

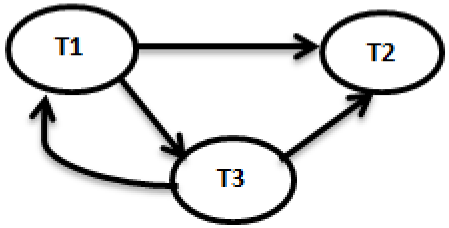
T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

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

Conflict Serializable??

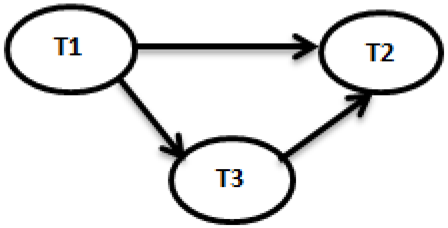
T1	T2	T3
R(X)		
		R(Z)
		W(Z)
R(Y)		
	R(Y)	
	W(Y)	
		W(X)
	W(Z)	
W(X)		



Non Conflict Serializable Schedule

Conflict Serializable??

T1	T2	T3
R(X)		
	R(Y)	
		R(Y)
	W(Y)	
W(X)		
		W(X)
	R(X)	
	W(X)	



T1→T3→T2

Conflict Serializable Schedule

Conflict Serializable??

S: R1 (B), R3(C), R1 (A), W2(A), W1(A), W2(B), W3 (A), W1 (B), W3 (B), W3(C)

T1	T2	T3
R(B)		
		R(C)
R(A)		
	W(A)	
W(A)		
	W(B)	
		W(A)
W(B)		
		W(B)
		W(C)

References

- 7th edition (Chapter 17)