Transaction

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items
  - A transaction is the DBMS's abstract view of a user program: a sequence of reads and writes
- A transaction must see a consistent database
- During transaction execution the database may be temporarily inconsistent
  - A sequence of many actions which are considered to be one atomic unit of work
- When the transaction completes successfully (is committed), the database must be consistent
  - After a transaction commits, the changes it has made to the database persist, even if there are system failures
- Multiple transactions can execute in parallel
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties

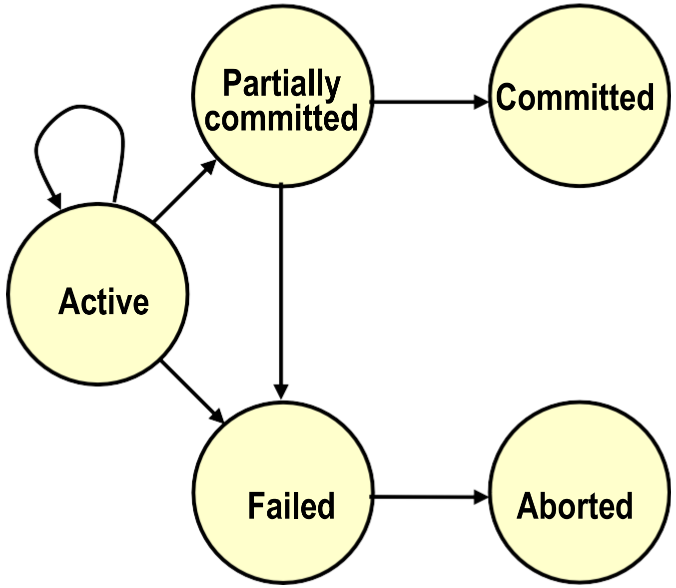
- To preserve the integrity of data the database system transaction mechanism must ensure:
  - ***Atomicity***. Either all operations of the transaction are properly reflected in the database or none are
  - ***Consistency***. Execution of a transaction in isolation preserves the consistency of the database
  - ***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
    - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished
  - ***Durability***. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

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*)
  - **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.
  - **Consistency requirement** – the sum of A and B is unchanged by the execution of the transaction.
  - **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)
    - Isolation can be ensured trivially by running transactions **serially**, that is one after the other.
    - However, executing multiple transactions concurrently has significant benefits in DBMS throughput
  - **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.

Transaction States

- **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; can be done only
    - if no internal logical error occurred
    - Kill the transaction
- **Committed**
  - after successful completion



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; that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Concurrency Control

- Concurrency control (CC) is a process to ensure that data is updated correctly and appropriately when multiple transactions are concurrently executed in DBMS
- Obviously some form of concurrency control mechanism is necessary to enable transactions to run concurrently as far as possible; but controlled in such a way that the effect is the same as if they had been run serially.

What Happen in a Transaction

- Retrieve : ‘read’ (R)
- Update : ‘write’ (W).

interleaving two transactions => 3 PBS:

RR – no problem

WW – lost update

WR – uncommitted dependency

RW – inconsistent analysis

Three classic problems

Although transactions execute correctly, results may **interleave** in diff ways =>

3 classic problems.

- Lost Update
- Uncommitted Dependency
- Inconsistent Analysis

Lost Update problem

Time	User 1 (Trans A)	User2 (Trans B)
1	Retrieve t= 40	
2		Retrieve t= 40
3	Update t t=40+10 =50	
4		Update t t= 40-10 = 30
5		
6		

t is a tuple in a table retrieved by both users in the course of both transactions. Transaction A loses an update at time 4. The update at t3 by transaction A is lost (overwritten) at t4 by B.

Uncommitted Dependency

Time	User 1 (Trans A)	User 2 (Trans B)
1		Retrieve t = 40 Update t = 50
2	Retrieve t = 50	
3		Rollback t= 40
4		
5	Update t= 50+10 = 60	
6		

One trans is allowed to retrieve/update) a tuple updated by another, but **not yet committed**.
Trans A is dependent at time t2 on an uncommitted change made by Trans B, which is lost on Rollback.

Inconsistent Analysis

Initially: Acc 1 = 40; Acc2 = 50; Acc3 = 30;

Trans A sees **inconsistent DB state** after B updated Accumulator

=> performs inconsistent analysis.

Time	User 1 (Trans A)	User 2 (Trans B)
1	Retrieve Acc 1: Sum = 40	
2	Retrieve Acc 2: Sum = 90	
3		Retrieve Acc3 :
4		Update Acc3: 30 → 20
5		Retrieve Acc1:
6		Update Acc1: 40 → 50
7		commit
8	Retrieve Acc3: Sum = 110 (not 120)	