Unit-4

PART-I TRANSACTION PROCESSING CONCEPT

Transaction

→ A transaction can be defined as a group of tasks. A single task is the minimum processing unit which can not be divided further.

→ Example:

Suppose we have to let's take an example of a simple transaction.
Suppose a bank employee transfer Rs 500 from A's account to B's account. This very simple & small transaction involves several low level tasks.

A's Account	B's Account
Open-Account(A)	Open-Account(B)
Old-Balance = A.balance	Old-Balance = B.balance
New-Balance = Old-Balance - 500	New-Balance = Old-Balance + 500
A.balance = New-Balance.	B.balance = New-Balance
Close-Account(A)	Close-Account(B)

→ ACID Properties

* A transaction is a very small unit of a program & it may contain several low level tasks.

* A transaction in a database system must maintain Atomicity, Consistency, Isolation & Durability - Commonly known as ACID properties in order to ensure accuracy, Completeness & data Integrity.

1. **Atomicity**

* This property states that a transaction must be treated as a atomic unit.
* Atomic Transaction means either all operations within a transaction are completed or none.
* There must be no states in ~~trans~~ database where a transaction is left partially completed

2. **Consistency** * The database must remain in a consistent state after any transaction.

* A transaction must be executed in Isolation.
* It means variables used by a transaction cannot be changed by any other transaction concurrently.
* If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

4. **Durability** * The database should be durable enough to hold all its latest updates even if system fail or restarts.

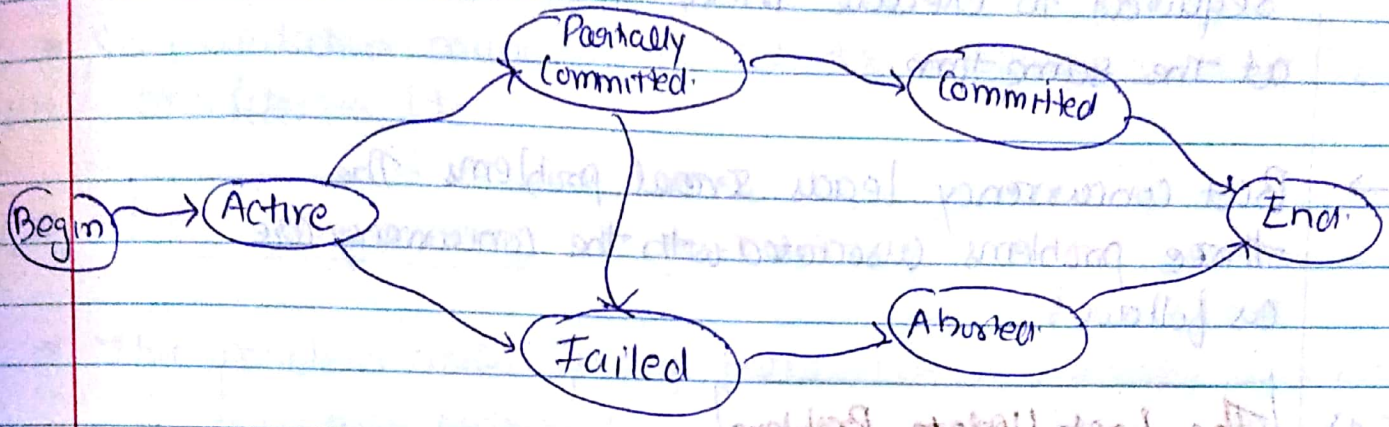
* Changes are made permanent to database after after successful completion of transaction even in the case of system failure or crash.

3. **Isolation**

* In a ~~transaction~~ database system where more than one transaction are being executed simultaneously & in parallel, the property of Isolation states that all the transactions will be carried out & executed as if it is the only transaction in the system.
* No transaction will affect the existence of any other transaction.

→ **Transaction States**

A transaction must be in one of the following states:



(i) **Active state**: In this state, transaction is being executed. This is the initial state of the transaction.

(ii) **Partially Committed**: When a transaction executes its final operation, it is said to be in a partially committed state.

(iii) **Failed**: In any case, if transaction cannot be proceeded further then transaction is in failed state.

(iv) **Committed**: After successful completion of transaction, it is in committed state.

(v) **Aborted**: If any of the checks fails & the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to the original state where it was prior to the execution of the transaction. Transaction in this state are called aborted.

Why Concurrency Control is Needed :

→ To make system efficient & save time, it is required to execute more than one transaction at the same time.

→ But concurrency leads several problems. The three problems associated with the concurrency are as follows :

(1) The Lost Update Problem

* If any transaction T_j updates any variable v at time t without knowing the value of v at time t then this may leads to lost update problem

Consider the transactions shown in fig.

Transaction T_i	Time	Transaction T_j
–	↓	–
read (v)	t_1	–
–	↓	–
–	t_2	read (v)
–	↓	–
update (v)	t_3	–
–	↓	–
–	t_4	update (v)
–	↓	

* At time t_1 & t_2, transactions t_i & t_j reads variable v respectively & gets same value of v.

* At time t_3, t_i updates but at t_4, t_j also updates v (old value of v) without looking the new value of v (updated by t_j)

* So, updation made by t_i at t_3 is lost at t_4 bec. T_j will overwrites it.

(2) The Uncommitted Dependency Problem

* This problem arises if any transaction T_j updates any variable v & allows retrieval or updation of v by any other transaction but T_j rolled back due to failure.

* Consider the transaction shown in fig.

Transaction T_i	Time	Transaction T_j
–	t_1	–
–		write (v)
–	t_2	–
read(v) or write(v)	t_3	–
–		–
–		Rollback
–		

* At time t_3, transaction T_j updates variable v which is read or updated by T_i at times t_2.

* Suppose at time t_3, T_j is ~~rolled~~ rollbacked due to any reason then result produced by T_i is wrong. because it is based on the false assumption.

(3) The Inconsistent Analysis Problem

* Consider the transactions shown in fig.

Transaction T_i	Time	Transaction T_j
—	↓	—
read(a) a=10	t_1	—
read(b) b=20	t_2	—
—	↓	
—	t_3	Write(a) a=50
—	↓	—
—	t_4	Commit
—	↓	—
—	t_5	
Add a, b		
a+b=30, not 60	↓	
—		

* the transaction T_i reads variable a & b at time t_1 & t_2 respectively

* But at time t_3, T_j updates value of a from 10 to 50 & commits at t_4 that makes changes permanently

* So, addition of a & b at time t_5 give wrong result.

* This leads inconsistency in database bec of inconsistent analysis by T_i.

Concurrency Techniques

→ To avoid concurrency related problems & to maintain consistency, some protocols need to be made so that system can be protected from such situations & inc. the efficiency.

→ We have concurrency control protocols to ensure atomicity, isolation & serializability transactions.

→ Concurrency control protocols can be broadly divided into 2-categories:

(1) Lock-Based Protocols (2) Time Stamp-Based protocols.

→ (1) Lock-Based Protocols

* Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write until it acquires an appropriate lock on it.

* There are two types of Locks:

(i) Shared Lock: • Shared lock is Read-only lock.
• if a transaction Ti has obtained shared lock on data item A then Ti can read A but cannot modify A.
• It is denoted by S & Ti is said to be shared lock mode.

(ii) Exclusive Lock: • Exclusive lock is Read-write lock.
• if a transaction Ti has obtained exclusive lock on data item A then Ti can both read & modify (write) A.
• It is denoted by X & Ti is said to be exclusive lock mode.

* Compatible lock modes:

S-lock (A) - Shared lock on data item A
X-lock (A) - Exclusive lock on data item A
read (A) - Read operation on A
write (A) - write operation on A.
unlock (A) - A is free now.

* Example:

- Consider the banking system in which you have to transfer Rs 500/- from account A to B.
- Transaction T_1 in fig(a) shows transfer of amount & T_2 shows sum of account A & B
- Initially account A has Rs 1000/- & B has 300/-.

T_1	T_2
X-lock (A);	S-lock (A)
read (A);	read (A)
A = A - 500;	unlock (A);
write (A);	S-lock (B)
Unlock (A);	read (B)
X-lock (B);	unlock (B);
read (B);	display (A+B);
B = B + 500;	
Write (B);	
Unlock (B);	

fig(a)

Transaction T_1 & T_2 for banking example

- The possible schedule for T_1 & T_2 is shown below:

~~PART-2~~ ~~Concurrency Control~~

T_1		T_2	
X-lock (A)			
read (A)	A = 1000		
A = A - 500	A = 500		
write (A)			
unlock (A)			
		S-lock (A)	
		read (A)	A = 500
		unlock (A)	
		S-lock (B)	
		read (B)	B = 300
		unlock (B)	
		display (A+B)	A+B = 800
X-lock (B)			
read (B)	B = 300		
B = B + 500	B = 800		
write (B)			
unlock (B)			

fig(b) (Schedule for transaction T_1 & T_2 in case locked data items are unlocked immediately after its final access)

- In the schedule shown in fig(b), T_2 gives wrong result i.e. Rs 800/- instead of Rs 1300/- bec. unlock A is performed immediately after its final access.

* Improvement in Basic Structure of Locking:

Keep all locks until the end of transaction.

The modified transaction T_1 & T_2 are shown below

T_1	T_2
X-lock(A);	S-lock(A);
read(A);	read(A);
A = A - 500;	
write(A);	S-lock(B);
X-lock(B);	read(B);
read(B);	display(A+B);
B = B + 500;	unlock(A);
write(B);	unlock(B);
unlock(A);	
unlock(B);	

(Improved structure of locking on transaction)

- The transaction T_2 cannot obtain lock on A until A is released by T_1 but at that time changes are made in both accounts.

- It maintains serializability & solve all concurrency problems.
- But it leads to a new problem that is Deadlock.

* Two-phase locking protocol:

- Two-phase locking protocol guarantees serializability.
- It has two phases:

(i) Growing Phase - transaction can obtain new locks but can not release any lock.

(ii) Shrinking Phase - transaction can release locks but cannot obtain new locks.

* Lock Point : the point at which transaction has obtained its final lock.

- types :

(i) Basic two phase locking protocol

> It ensures serializability
> It may cause deadlock.
> It may cause cascading rollback.

(ii) Strict two phase locking protocol

> In this all exclusive locks are kept until the end of transaction or until the transaction commits.
> there is no cascading rollback.

S1.

T_1	T_2
S-lock(A)	
read(A)	
X-lock(B)	
read(B)	
write(B)	
X-lock(C)	
unlock(B)	
read(C)	X-lock(B)
write(C)	read(B)
–	write(B)
–	–
–	–
–	–
Rollback	Rollback due to rollback of T_1

S2.

T_1	T_2
S-lock(A)	
read(A)	
X-lock(B)	
read(B)	
write(B)	
X-lock(C)	
read(C)	
write(C)	
commit	
unlock(B)	
unlock(A)	X-lock(B)
unlock(C)	read(B)
	write(B)
	–
	–

(2) Timestamp - Based Protocols

* The most commonly used concurrency protocol is the timestamp based protocol.
* This protocol uses either system time or logical counter as a timestamp.
* Lock-based protocols manage the order b/w the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as transaction is created.

* Timestamp ordering Protocol

- the timestamp-ordering protocol ensures serializability among transactions in their conflicting read & write operations.
- This is the responsibility of the protocol system that the conflicting pair of tasks should be executed acc. to the timestamp values of the transactions.
  - The timestamp of transaction Ti is denoted as TS(Ti)
  - Read time-stamp of data-item X is denoted by R-timestamp(X).
  - Write time-stamp of data-item X is denoted by W-timestamp(X).
- Timestamp ordering protocol works as follows :-

(i) If a transaction Ti issues a read(X) operation -

The

- If TS(Ti) < W-timestamp(X) : operation rejected
- if TS(Ti) >= W-timestamp(X) : operation executed
- All data-item timestamps updated

(ii) if a transaction Ti issues a write(X) operation :-

- if TS(Ti) < R-timestamp(X) : Operation rejected.
- if TS(Ti) < W-timestamp(X) : operation rejected & Ti rolled back
- Otherwise operation executed

- Advantages

(i) It maintain serializability (iii) It is free from cascading rollbacks.
(ii) It is free from deadlock.

- Disadv :

(i) Starvation is possible for long transactions
(ii) Schedules are not recoverable.

Deadlocks

→ A system is said to be in deadlock state if there exist a set of transactions {T0, T1, ... Tn} such that each transaction in set is waiting for releasing any resources by any other transaction in that ~~state~~ set

→ Necessary conditions for Deadlock : A system is in deadlock state if it satisfies all of the following conditions.

(i) Hold & wait : whenever a transaction holding at least one resource & waiting for another resources that are currently being held by other processes.

(ii) Mutual Exclusion: When any transaction has obtained a nonsharable lock on any data item & any other transaction requests that item.

(iii) No Preemption: When a transaction holds any resource even after its completion.

(iv) Circular wait: let T be the set of waiting processes $T = \{T_0, T_1, \ldots, T_i\}$ such that T_i is waiting for a resource that is held by T_0.

→ **Methods for handling Deadlocks:**

There are two methods for handling deadlocks These are -

(1) **Deadlock Prevention:**

* It is better to prevent the system from deadlock condition then to detect it & then recover it.
* Diff. approaches for preventing deadlocks are -

(i) **Advance Locking**

* It is the simplest technique in which each transaction locks all the required data items at the beginning of its execution in atomic no.
* It reduces concurrency.

(ii) **Ordering Data Items**

* In this approach, all the data items are assigned in order such as in ascending or descending order

(ii) It reduces concurrency but less than advance locking.

(iii) **Ordering Data items with Two-phase locking** :

* In this approach, two phase locking with ordering data items is used to inc concurrency.

(iv) **Techniques based on timestamps** : There are two diff techniques to remove deadlock based on timestamps -

* **Wait-die** : if $TS(T_i) < TS(T_j)$
  T_i waits;
  else
  T_i is rolled back;
  It is a non-premetive technique.

* **Wound-wait** : if $TS(T_i) > TS(T_j)$
  T_j waits;
  else
  T_j is rolled back;
  It is a premetive technique.

(2) **Deadlock Detection & Recovery.**

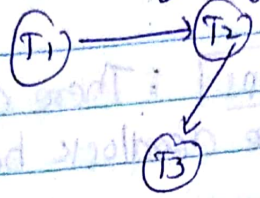
* In this approach, allow system to enter in deadlock state then detect it & recover it.
* To ~~employ~~ Satisfy this approach, system must have :

(a) Mechanism to maintain info. of current allocation of data items to transactions & the order in which they are allocated.
(b) An algo. which is invoked periodically by system to determine deadlocks in system.
(c) An algo. to recover from deadlock state.

* Diff. approaches for deadlock detection are -

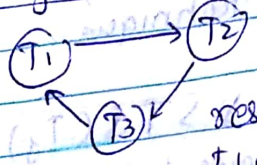
(i) **Wait-for Graph**: It consists of a pair $G=(V,E)$ where V is the set of vertices & E is the set of edges.

Consider a wait for graph as shown in fig



there are 3 transactions T_1, T_2, T_3 & there is no deadlock bec. there is no cycle in wait for graph

Consider a graph:



Thus there exist a cycle, $T1 \rightarrow T2 \rightarrow T3 \rightarrow T1$ which results in deadlock state & T_1, T_2, T_3 are blocked.

(ii) **Recovery from Deadlock**: When deadlock detection algo. detects any deadlock in system, system must take necessary actions to recover from deadlock.

Steps of recovery algo are:

(a) **Select a Victim**: The transaction which is rolled back is called known as victim.

(b) **Rollback**: After selection of victim, it is rollbacked.

(c) **Prevent starvation**: ~~Some times a particular transaction is rollbacked several times & never completes which causes starvation~~ To prevent starvation, set the limit of rollbacks, the same transaction chooses as victim. & many more.

Serializability

Topics
(i) Serializability
(ii) Serial schedules
(iii) Non-serial schedules
(iv) Conflict serializability.

Serializability

→ Consider a set of transactions $(T_1, T_2, \ldots, T_i)$.
→ S_1 is the state of database after they are concurrently executed & successfully completed.
→ And S_2 is the state of database after they are executed in any serial manner & successfully completed.
→ if S_1 & S_2 are same then database maintains serializability

↳ **Schedules** : * A chronological execution sequence of a transaction is called a schedule.
* A schedule can have many transactions in it, each comprising of a no of instructions / tasks.

(i) Serial schedules

* A serial schedule is a sequence of operation by a set of concurrent transactions that preserves the order of operation in each of the individual transactions.

* Transactions are performed in serial manner.

* No interference b/w transactions.

* It does not matter which transaction is executed first as long as every transaction is executed in its entirety from beg. to end.

* A serial schedule give the benefits of concurrent execution without any problem.
* Serial schedule does not interleaved the actions of diff transactions.
* Ex:
  if some transaction T is long, the other transaction must wait for T to complete all its operations.

(ii) Non-serial schedules

* A non-serial schedule is a schedule where the operations of a group of concurrent transactions are interleaved.
* Transactions are performed in non-serial ~~manner~~ order, but result should be same as serial.
* Concurrency problem can arise here.
* The problem ~~we have seen earlier~~ lost update, uncommited data, inconsistent analysis arise if schedule is not proper.
* Ex:
  In this schedule, the execution of other transaction goes on without waiting the completion of T.
* the objective behind the serializability is to find the non-serial schedule that allow transactions to execute concurrently without interfering one another.

Conflict serializability

* A schedule is conflict serializable if it is conflict equivalent to some serial schedule

Ex:

(a) T_1 T_2

R(A) R(A) → non conflicting pairs.

(b)

T_1	T_2
R(A)	
	W(A)

conflict pair.

(c)

T_1	T_2
W(A)	
	W(A)

conflict.

(d)

T_1	T_2
W(A)	
	R(A)

conflict.

* Pairs of operations are conflicting if:
  (a) At least one write operation
  (b) Both operations are done on same data item
  (c) Diff. transaction

* Testing for conflict serializability of schedules

Create a Precedence Graph G = (V, E)
V = ($T_1, T_2, \ldots T_n$) set of transaction.
E = ($E_1, E_2, \ldots E_n$) precedence.

(i) T_j executes a R(X) after T_i executes W(X)
↳ Create a edge ($T_i \rightarrow T_j$) in Precendence graph.

(ii) T_j executes a W(X) after T_i executes R(X)
↳ Create a edge ($T_i \rightarrow T_j$) in precedence graph.

(iii) T_j executes W(X) after T_i executes W(X)
↳ Create a edge ($T_i \rightarrow T_j$) in precedence graph

* Testing condition:

** If precedence graph is acyclic then schedule is ~~not~~ conflict serializable schedule.

→ If precedence graph is cyclic then schedule is not conflict serializable Schedule