DATE | | PAGE

Transaction:

unit of program execution that accesses and possibly updates various data-items.

- Atomicity
- Consistency
- Isolation
- Durability.

Transaction acess data using two operations:

read(x),
which transfers the data item X from database to a local buffer belonging to the transaction that executed the read operation

write(x)
transfers the data item X from the local buffer of transaction that executed the write back to the database.

ex: Let T_i be a transaction that transfers $50 from Account A to B

```
Ti : read (A)
     A := A-50;
     write (A);
     read (B)
     B := B+50;
     write (B);
```

1. Consistency:

Consistency requirement here is that sum of A and B be unchanged by the execution of transaction

- if database is consistant before execution of transac. database remains consistant after execution.
- responsibility of Application programmer

2. Atomicity.

failure:
power failure,
n/w failure,
s/w errors.

- Either all operations of the transactions are reflected properly in database, or none are.
- because of failure, the state of system no longer reflects correct state that the database is supposed to capture. that state is inconsistent. state.'
- Ensuring Atomicity is reponsibility of database System itself, it is handled by componant called transaction management componant.

3. Durability -

After a transaction completes successfully, the changes it had made to the database persist, even if system failure.

- failure of computer system may result in loss of data in main memory, but data written on disk never lost.

We can guarantee durability by ensuring that either

1. the updates carried out by transaction have been written to disk before the transaction completes.

2. information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when system is restarted after failure.

responsibility of a software conponant of DB system - recovery management Component.

- Isolation:

Even though multiple transactions may execute concurrently, the system guarantees that for every pair of transaction T_i & 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.

Thus transaction is unaware of other transactions executing concurrently in a system.

A way to avoid the problem of concurrently executing transaction is to execute transaction serially: that is one after other.

- responsibility of concurrency-control componant of DB system

- Transaction State:

In the absence of failure, all transaction complete successfully. - commited

- transaction may not always complete its execution successfully. Such a transaction is termed as aborted.

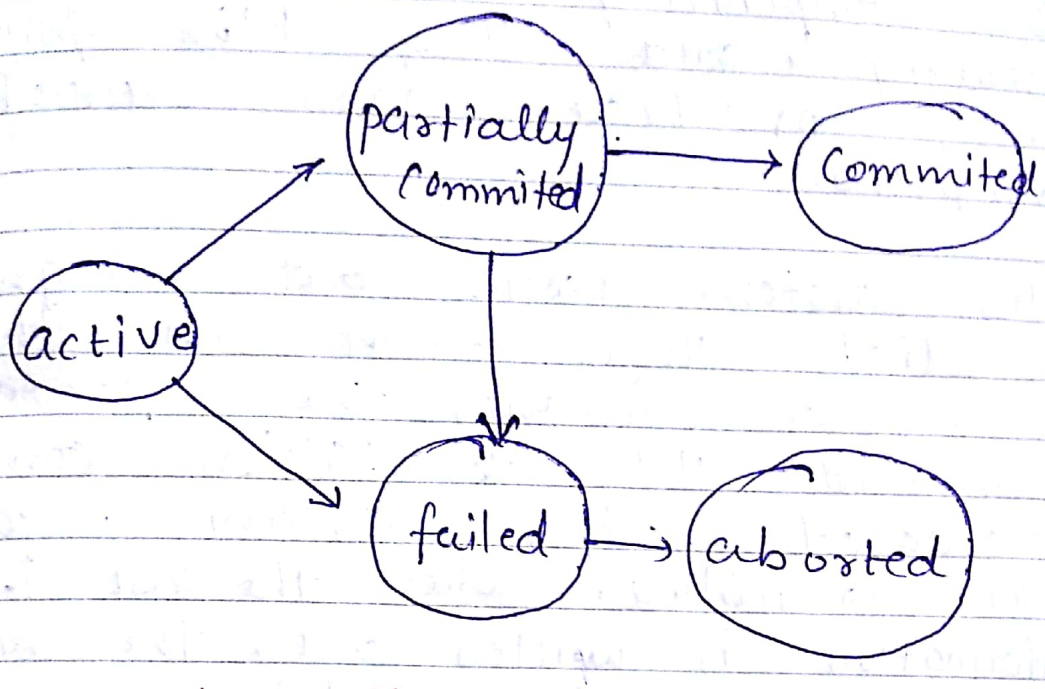
1. Active:
   initial state.
   - transaction stays in this state while it is executing

2. Partially Committed:
   after final state has been executed

3. failed:
   after discovery that normal execution can no longer proceed.

4. aborted:
   after transaction has been rolled back & database has been restored to its prior to the start of transaction.

5. Commited: after successful completion



state diagram of transaction

Compansating Transaction:

Once, a transaction has commited, we can't undo its effects by aborting it. The only way to undo the effects of commited transaction is to executing compansating transaction.

- transaction is either commited or aborted ⇒ terminated.

→ transaction starts in active state when it finishes its final statement, it enters the partially committed state.

At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted.

since the actual o/p may still be temporarily residing in main memory, and thus h/w failure may preclude its successfully completion.

→ DB system writes out enough info to disk that, even in the event of failure, the updates performed by transaction can be recreated when system restarts after a failure, when the last ~~instruction~~ information is written out, the transaction enters in commited state.

→ the transaction enters in failed state after the system determines that the transaction can no longer proceed with its normal execution. Such transacti must be rolled back. Then, it enters the aborted state.

At this time transaction has two options.

- restart the transaction.
- kill the transaction.

* Concurrent Executions

Transaction processing system allows multiple transactions to run concurrently.

→ allowing multiple transactions to update data concurrently causes several complications with consistency of data.

Ensuring concurrency in spite of concurrent execution requires extra work; it is far easier to insist that transactions run serially.

But, there are two good reasons to allow concurrency.

1. Improved throughput & resource utilizatio
2. Reduced waiting time

Throughput: number of transactions executed in a given amount of time.

→ processor & disk utilization also increased

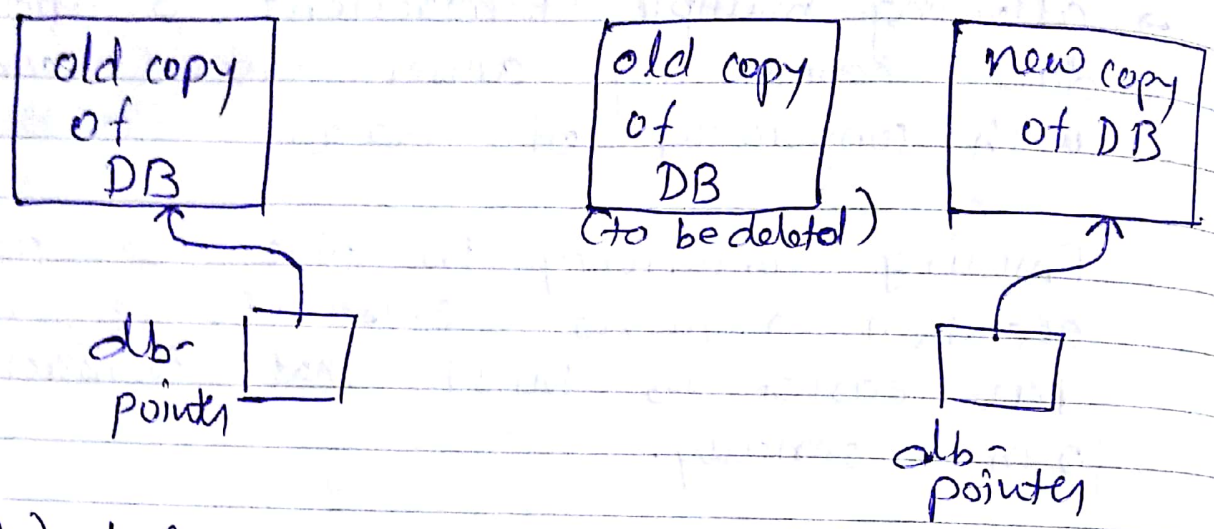
↳ mix transactions that run on system, some short & some long
if serially: short transaction has to wait for preceding long trans.

so, average response time will be reduced

DATE

Implementation of atomicity & Durability.

shadow copy:



(a) before update

(b) after update

This scheme is based on making copies of the database, called shadow copies, assumes that only one transaction is active at a time.

A pointer called db-pointer is maintained on disk; it points to the current copy of database.

In this scheme, the transaction that wants to update the database first creates a complete copy of database. All updates are done on new database copy, leaving the original copy - the shadow copy, untouched.

DATE | | PAGE

if transaction aborted ⇒ system deletes new copy.
old copy of db has not been affected.

If transaction completes, OS makes sure that all puges of new copy have been written out to disk.

→ After OS has written all puges to disk, database updates the pointer db-pointer to point new copy of the database; new copy they becomes the current copy of DB.

→ old copy is then deleted.

→ transaction is said to have commited at the point where updated db-pointer written to disk.

→ extremely insufficient in context of large DB, since executing a single transaction requires copy of entire DB.

- ~~Concurrent~~ Execution.

Schedule:

The execution sequence of transactions are called as schedule. They represent chronological order in which instructions are executed in system.

-

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

Schedule 1.

Serial schedule.

- Each serial schedule consists of sequence of instructions from various transactions.

For set of n transactions, n! different valid schedules.

DATE | | PAGE

When DB system executes several transactions concurrently, the corresponding schedule no longer needs to be serial.

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 : 2

- Concurrent schedule equivalent to schedule 1.

// task

Concurrency - control component.

Serializability.

Conflict Serializability
View Serializability.

1. Conflict Serializability:

If a schedule S can be transformed to S' by a series of swaps of non-conflicting instructions, we say that S & S' are conflict equivalent.

Let us consider schedule S in which there are two consecutive instructions, I_i & I_j, of transaction T_i & T_j respectively.

→ If I_i & I_j refers to different data items, we can swap.

but if I_i & I_j ~~the~~ refer to same data items, then order matters.

1. I_i = read (Q), I_j = read (Q)
   order does not matter

2. I_i = read (Q), I_j = write (Q)
   order matters.

3. write read
   matters

4. write write
   matters.

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 (B)	
	write (A)
	read (B)
	write (B)

Conflict equivalent.

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

Conflict Serializable

Schedule S is conflict Serializable if it is conflict equivalent to the serial schedule.

T_3	T_4
r (Q)	
	w (Q)
w (Q)	

not conflict Serializable

2. View Serializability.

View equivalent if these conditions are met.

1. For each data item Q, if transaction Ti reads initial value of Q in Schedule S, then Ti must in S' also read initial value of Q.

2. if Ti executes read(Q) in Schedule S, and if that value was produced by write(Q) operation executed by Tj, then read(Q) operation of Ti must in schedule S', also read the value of Q that was produced by same write(Q) opn.

3. For each data item Q, the transaction performs final write(Q) opn in S must perform final write(Q) in S'.

Schedule S is View serializable if it is view equivalent to serial schedule

ex:1)

S1

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
w(B)	
	r(B)
	w(B)

S2

T_1	T_2
r(A)	
w(A)	
r(B)	
w(B)	
	r(A)
	w(A)
	r(B)
	w(B)

	A		B		
S_1	T_1	T_2	T_1	T_2	View serializable
S_2	T_1	T_2	T_1	T_2	

(2)

S1

T_1	T_2
r(A)	
	w(A)
W(A)	

S2

T_1	T_2
r(A)	
W(A)	
	w(A)

	A	
S_1	T_1	T_1
S_2	T_1	T_2

$\Rightarrow$ not a view serializable

T_1	T_2
	w(A)

A

S_1 T_1 T_2

Recoverable Schedule:

where for each pair of transactions T_i & T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before commit of T_j.

T_1	T_2
R(A)	
A := A-50	
W(A)	
	R(A)
	X = A * 0.04
	A = A + X
	W(A)
	Commit
R(B)	
B = B+50	
W(B)	
Commit.	

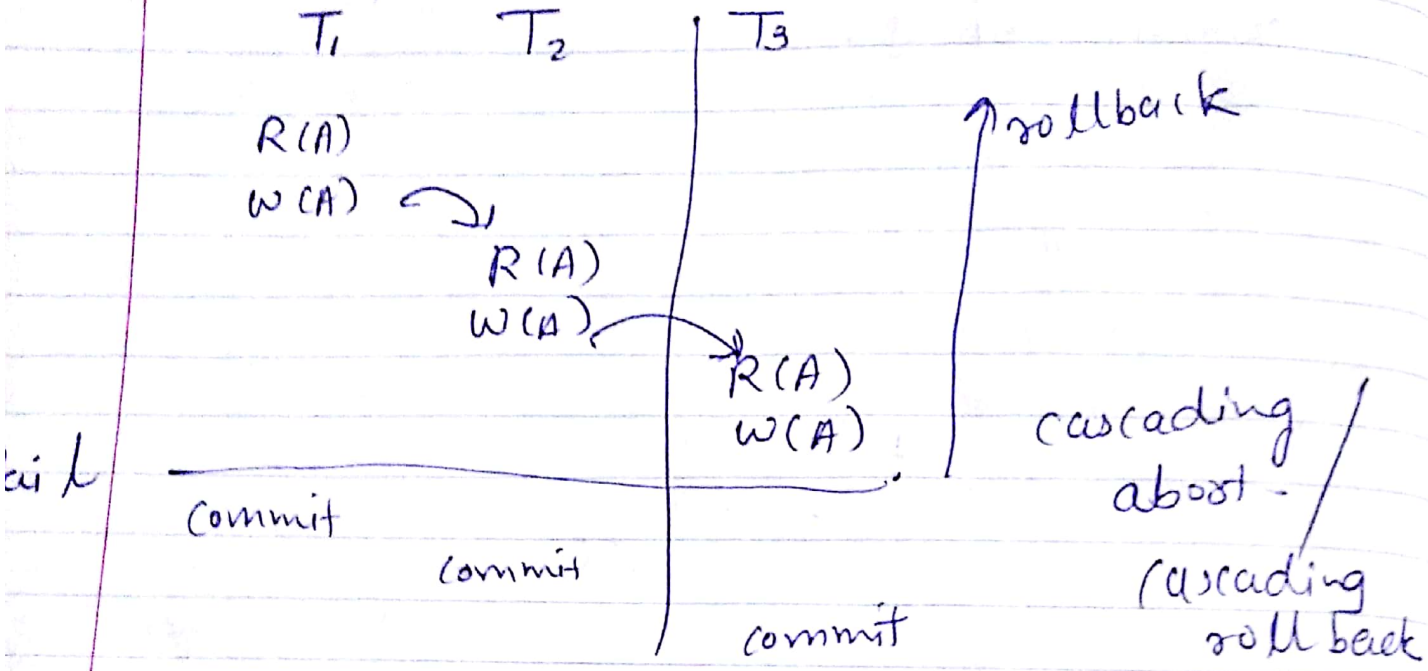
Roll back can't possible because T_2 is committed.

T_1	T_2
R(A)	
A = A-50	
W(A)	
	R(A)
	X = A * 0.04
	A = A + X
	W(A)
R(B)	
B = B+50	
W(B)	
Commit;	
	Commit;

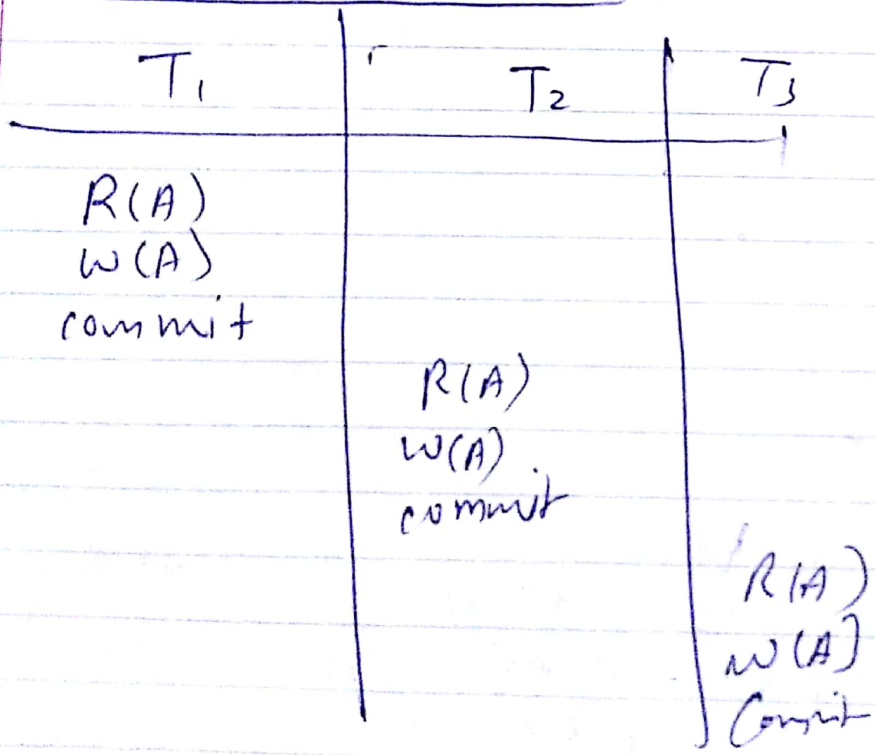
Roll backed.

Cascadeless Schedules.

Cascading aborts:



Cascadeless schedule



Cascadeless is always recoverable.

Cascadeless schedule is one where, for each pair of transaction T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read of T_j.

→ cascadeless schedule is also recoverable

6 Implementation of Isolation:

conflict serializable
view serializable
cascadeless.
} consistent database.

→ various concurrency-control-schemes that we can use to ensure that, even multiple transactions are executed concurrently, only accaptable schedules are generated, regardless of how the OS time-shares resources among the transactions.

lock. on entire DB before it starts and releases the lock after it has commited

∴ Only one transaction can execute at a time.