Transaction Management

Vu Tuyet Trinh
trinhvt@soict.hust.edu.vn

Department of Information Systems
SoICT-HUST

Transaction – example

500USD

Account A → **Account B**

read(A)

If A > 500 then

B:=B+500

A:=A-500

Crash

What happen ???

Transaction

- A sequence of read and write operations on data items that logically functions as one unit of work
  - Assuring data integrity and correction

- ACID Properties
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

Concurrency Control

Recovery

Automicity

- guarantee that either all of the tasks of a transaction are performed or none of them are

- Example

```
T:  Read(A,t1);
    If t1 > 500 {
          Read(B,t2);
          t2:=t2+500;        crash
          Write(B,t2);
          t1:=t1-500;
          Write(A,t1);
    }
```

Consistency

- ensures that the DB remains in a consistent state before the start of the transaction and after the transaction is over
- Example

```
T: Read(A,t1);              <-------  A+B = C
   If t1 > 500 {
       Read(B,t2);
       t2:=t2+500;
       Write(B,t2);
       t1:=t1-500;
       Write(A,t1);         <-------  A+B = C
   }
```


Isolation

- ability of the application to make operations in a transaction appear isolated from all other operations.
- Example A= 5000, B= 3000

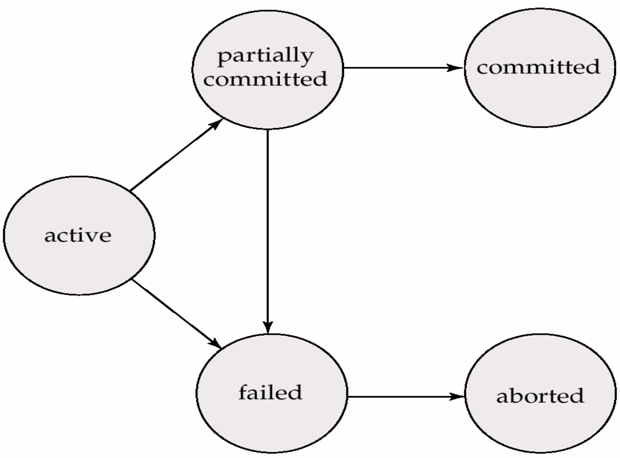
```
T: Read(A,t1);
   If t1 > 500 {
       Read(B,t2);
       t2:=t2+500;
       Write(B,t2);         <-------  T': A+B
                                      (= 5000+3500)
       t1:=t1-500;
       Write(A,t1);         <-------  (A+B = 4500+3500)
   }
```

Durability

- guarantee that once the user has been notified of success, the transaction will persist, and not be undone
- Ví dụ: A= 5000, B= 3000

```
T:  Read(A,t1);
    If t1 > 500 {
            Read(B,t2);
            t2:=t2+500;
            Write(B,t2);
            t1:=t1-500;
            Write(A,t1);
    }                          <------ crash
                                  A= 4500, B=3500
```


Transaction States



Transaction Management Interfaces

- Begin Trans
- Commit ()
- Abort()

- Savepoint Save()
- Rollback (savepoint)
  (savepoint = 0 ==> Abort)

Concurrency Control

- Objective:
  - ensures that database transactions are performed concurrently without the concurrency violating the data integrity
  - guarantees that no effect of committed transactions is lost, and no effect of aborted (rolled back) transactions remains in the related database.
- Example

T0: read(A);
A := A -50;
write(A);
read(B);
B := B + 50;
write(B);
	T1: read(A);
	temp := A *0.1;
	A := A -temp;
	write(A);
	read(B);
	B := B + temp;
	write(B);

Scheduling

T_0	T_1
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)

(1)

T_0	T_1
	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)	

(2)

T_0	T_1
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)

(3)

Serializability

- A schedule of a set of transactions is a linear ordering of their actions
  - e.g. for the simultaneous deposits example:

    R1(X) R2(X) W1(X) W2(X)

- A serial schedule is one in which all the steps of each transaction occur consecutively

- A serializable schedule is one which is equivalent to some serial schedule

Lock

- Definition
  - a synchronization mechanism for enforcing limits on access to DB in concurrent way.
  - one way of enforcing concurrency control policies
- Lock types
  - Shared lock (LS) readable but can not write
  - Exclusive lock (LX): read and write
  - UN(D): unlock
- Compatibility

	LS	LX
LS	true	false
LX	false	false

Example

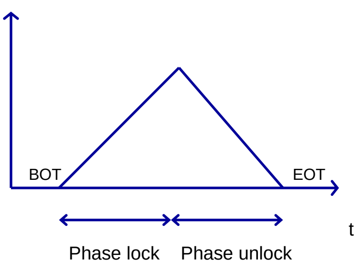
T0:	T1:
LX(A);	LX(A);
read(A);	read(A);
A := A -50;	temp := A *0.1;
write(A);	A := A -temp;
LX(B);	write(A)
read(B);	LX(B);
B := B + 50;	read(B);
write(B);	B:=B+temp;
UN(A);	write(B);
UN(B);	UN(A);
	UN(B);

Well-Formed, two-phased transaction

- A transaction is well-formed if it acquires at least a shared lock on Q before reading Q or an exclusive lock on Q before writing Q and doesn't release the lock until the action is performed
  - Locks are also released by the end of the transaction
- A transaction is two-phased if it never acquires a lock after unlocking one
  - i.e., there are two phases: a *growing phase* in which the transaction acquires locks, and a *shrinking phase* in which locks are released

2Phase Locking (2PL)

- Phase 1
  - locks are acquired and no locks are released
- Phase 2
  - locks are released and no locks are acquired



Example

T_1
Lock(A)
Read(A)
Lock(B)
Read(B)
B:=B+A
Write(B)
Unlock(A)
Unlock(B)

T_2
Lock(B)
Read(B)
Lock(A)
Read(A)
Unlock(B)
A:=A+B
Write(A)
Unlock(A)

2PL

T_3
Lock(B)
Read(B)
B=B-50
Write(B)
Unlock(B)
Lock(A)
Read(A)
A=A+50
Write(A)
Unlock(A)

T_4
Lock(A)
Read(A)
Unlock(A)
Lock(B)
Read(B)
Unlock(B)
Pritn(A+B)

Not 2PL

Deadlock

T0:		T1:	
LX(B);	(1)	LX(A);	(4)
read(B);	(2)	read(A);	(5)
B := B +50;	(3)	temp := A *0.1;	(6)
write(B);	(8)	A := A -temp;	(7)
LX(A);	(10)	write(A)	(9)
read(A);		LX(B);	
A := A - 50;		read(B);	
write(A);		B:=B+temp;	
UN(A);		write(B);	
UN(B);		UN(A);	
		UN(B);	

Resolving Deadlock

- Detecting
  - Recovery when deadlock happen
    - rollback
  - Used waiting-graph

- Avoiding
    - Resource ordering
    - Timeout
    - Wait-die
    - Wound-wait

Waiting Graph

- Graph
  - Node handling lock or waiting for lock
  - Edge T→U
    - U handle L(A)
    - T wait to lock A
    - T must wait until U unlock A

- If there exists a cycle in the waiting graph → deadlok

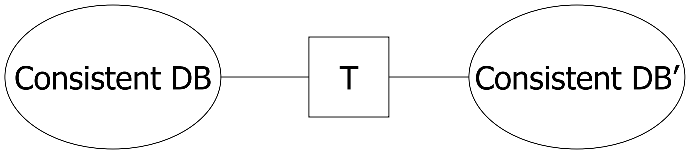
Timeout

- Set a limit time for each transaction
- If time-out → do rollback

Exercises

Transaction – consistency

collection of action that preserve consistency



with assumption

IF	T starts with consistent state + T executes in isolation
THEN	T leaves consistent state

How can constraints be violated?

- Transaction bug
- DBMS bug
- Hardware failure

  e.g., disk crash
- Data sharing

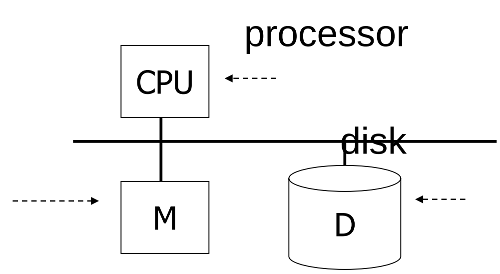
  e.g., T1 and T2 in parallel

Failures

Events — Desired

Undesired — Expected

Unexpected



Recovery

- Maintaining the consistency of DB by ROLLBACK to the last consistency state.
- Ensuring 2 properties
  - Atomic
  - Durability

➢ Using LOG

Transaction Log

- A sequence of log record keeping trace of actions executed by DBMS

  <start T>

    Log the beginning of the transaction execution

  <commit T>

    transaction is already finished

  <abort T>

    Transaction is calcel

  <T, X, v, w>

    Transaction makes an update actio, before update X=v, after update x = w

---

- Read(A)
- If A > 50 then display("so du hop le")
- Else {

    A:=A+50

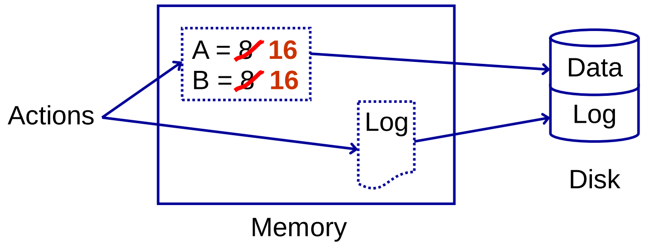
    ==========➔CRASH

    display ("ghi no tai khoan A")

  }

Transaction Log

- Handled in main memory and put to external memory (disk) when possible



Checkpoint

- Definition:
  - moment where intermediate results and a log record are saved to disk.
  - being initiated at specified intervals
- Objective
  - minimize the amount of time and effort wasted when restart
  - the process can be restarted from the latest checkpoint rather than from the beginning.
- Log record

  <checkpoint> or <ckpt>

Undo-logging

Step	Action	t	Mem A	Mem B	Disk A	Disk B	Mem Log
1							<start T>
2	Read(A,t)	8	8		8	8	
3	t:=t*2	16	8		8	8	
4	Write(A,t)	16	16		8	8	<T, A, 8>
5	Read(B,t)	8	16	8	8	8	
6	t:=t*2	16	16	8	8	8	
7	Write(B,t)	16	16	16	8	8	<T, B, 8>
8	**Flush log**						
9	Output(A)	16	16	16	16	8	
10	Output(B)	16	16	16	16	16	
11							<commit T>
12	**Flush log**						

Undo-Logging Rules

(1) For every action generate undo log record (containing old value)

(2) Before X is modified on disk, log records pertaining to X must be on disk (write ahead logging: WAL)

(3) Before commit is flushed to log, all writes of transaction must be reflected on disk

Undo Logging Recovery Rules

- Let S is set of unfinished transactions
  - <start T_i> in log
  - <commit T_i> or <abort T_i> is not in log
- For each <T_i, X, v> in log
  - If $T_i \in S$ then - Write(X, v)
    - Output(X)
- For each $T_i \in S$
  - Write <abort T_i> to log

Undo-Logging & Checkpoint

<start T_1>
<T_1, A, 5>
<start T_2>
<T_2, B, 10>
<T_2, C, 15>
<T_2, D, 20>
<commit T_1>
<commit T_2>
<checkpoint>
<start T_3>
<T_3, E, 25>
<T_3, F, 30>

scan

<start T_1>
<T_1, A, 5>
<start T_2>
<T_2, B, 10>
<start ckpt (T_1,T_2)>
<T_2, C, 15>
<start T_3>
<T_1, D, 20>
<commit T_1>
<T_3, E, 25>
<commit T_2>
<end ckpt>
<T_3, F, 30>

scan

Redo-logging

Step	Action	t	Mem A	Mem B	Disk A	Disk B	Mem Log
1							<start T>
2	Read(A,t)	8	8		8	8	
3	t:=t*2	16	8		8	8	
4	Write(A,t)	16	16		8	8	<T, A, 16>
5	Read(B,t)	8	16	8	8	8	
6	t:=t*2	16	16	8	8	8	
7	Write(B,t)	16	16	16	8	8	<T, B, 16>
8							<commit T>
9	**Flush log**						
10	Output(A)	16	16	16	16	8	
11	Output(B)	16	16	16	16	16	
							<T, end>

Redo-logging Rules

(1) For every action, generate redo log record (containing new value)

(2) Before X is modified on disk (DB),all log records for transaction that modified X (including commit) must be on disk

(3) Flush log at commit

(4) Write END record after DB updates flushed to disk

Redo-logging Recovery Rules

- Let S = set of transactions with
  - <Ti, commit> in log
  - no <Ti, end> in log
- For each <Ti, X, w> in log, in forward order (earliest → latest)
  - If Ti ∈ S then write(X, w)
    output(X)
- For each Ti ∈ S
  - write <Ti, end>

Redo Logging & Checkpoint

```
<start T1>
<T1, A, 5>
<start T2>
<commit T1>
<T2, B, 10>
<start ckpt (T2)>
<T2, C, 15>
<start T3>
<T3, D, 20>
```

scan

```
<start T1>
<T1, A, 5>
<start T2>
<commit T1>
<T2, B, 10>
<start ckpt (T2)>
<T2, C, 15>
<start T3>
<T3, D, 20>
<end ckpt>
<commit T2>
<commit T3>
```

scan

Discussion

- Undo Logging
  - need to write to disk as soon transaction finishes
  - Access disk
- Redo Logging
  - need to keep all modified blocks in memory until commit
  - Use memory

Undo/Redo Loggin

Step	Action	t	Mem A	Mem B	Disk A	Disk B	Mem Log
1							<start T>
2	Read(A,t)	8	8		8	8	
3	t:=t*2	16	8		8	8	
4	Write(A,t)	16	16		8	8	<T, A, 8, 16>
5	Read(B,t)	8	16	8	8	8	
6	t:=t*2	16	16	8	8	8	
7	Write(B,t)	16	16	16	8	8	<T, B, 8, 16>
8	**Flush log**						
9	Output(A)	16	16	16	16	8	
10							<commit T>
11	Output(B)	16	16	16	16	16	

Undo/Redo Logging Rules

- Page X can be flushed before or after T commit
- Log record flushed before corresponding updated page (WAL)
- Flush at commit (log only)

Undo/Redo Logging & Checkpoint

```
<start T1>
<T1, A, 4, 5>
<start T2>
<commit T1>
<T2, B, 9, 10>
<start ckpt (T2)>
<T2, C, 14, 15>
<start T3>
<T3, D, 19, 20>
<end ckpt>
<commit T2>
```

scan

```
<start T1>
<T1, A, 4, 5>
<start T2>
<commit T1>
<start T3>
<T2, B, 9, 10>
<T3, E, 6, 7>
<start ckpt (T2, T3)>
<T2, C, 14, 15>
<T3, D, 19, 20>
<end ckpt>
<commit T2>
```

scan

Undo/Redo Logging Recovery Rules

- Backwards pass (end of log → latest valid checkpoint start)
- Constructing set S of committed transactions
- undo actions of transactions not in S
- undo pending transactions
- follow undo chains for transactions in (checkpoint active list) – S
- Forward pass (latest checkpoint start → end of log)
- redo actions of S transactions

44

Isolation Levels

- Read Uncomitted (No lost update)
  - Exclusive locks for write operations are held for the duration of the transactions
  - No locks for read
- Read Committed (No inconsistent retrieval)
  - Shared locks are released as soon as the read operation terminates.
- Repeatable Read (no unrepeatable reads)
  - Strict two phase locking
- Serializable (no phantoms)
  - Table locking or index locking to avoid phantoms