

# Introduction

## Transaction Processing

- A very important component of any online information system / portal
- E-governance applications are in wide-spread use

## Transaction

- A logical unit of work to be carried out
- on request by the end-user
  - transfer of specified amount of money from one account to another
  - making a reservation for a journey
  - issue a book of the library to a user ....

# Assumptions

- DB Server is a single processor system
- 2-Tier architecture is used: DB Server - Client System
- Transactions considered in this module
  - Involve a single DB but not multiple DBs
  - Not nested – a Txn does not initiate another Txn inside it
  - Transactions do not exchange any messages
- E-commerce Transactions
  - Might involve multiple DBs – Merchant, Bank, etc
  - Additional issues need to be considered

# Transaction Processing

## Input:

A number of requests for services from the end-users submitted concurrently from several input points

## Action:

Carry out the requested services in a *consistent* manner

- no seat on a journey is reserved for more than one person
- amount debited from the party A is credited to party B

## Measure:

Maintain a reasonably high throughput

- number of transactions completed per sec

# What is a Transaction?

From the end-user point of view

- a logically sensible/complete piece of work

From the system point of view

- a sequence of database operations
  - read data from tables on the disk,
  - compute/make the updates,
  - write back the data to the disk

Doing one transaction at a time, one-by-one:

- no scope of errors, but slow

Doing multiple transactions at a time

- Scope for error unless done carefully, good throughput

## ACID Properties

- A - Atomicity
- C - Consistency
- I - Isolation
- D - Durability

Important properties to be satisfied by the overall system

## Atomicity

- The work of a transaction should be done entirely as one unit or not performed at all
  - Carrying out some portion of the work leads to inconsistent state of the database
    - For example, Rs.1000/-, to be transferred from A to B
    - Debited from A but not credited to B!
  - system encountered on error and stopped in the middle!
    - can not be allowed; leads to inconsistency
- If only a part of the work is done and error is encountered
  - ensure that effect of the partial work is not reflected in the database
- Responsibility of Recovery Module

# Consistency

- Correctness Assumption:
  - A transaction takes the system from one *consistent* state to another *consistent* state, if executed in *isolation*
  - Responsibility of the application program developer
    - application programs or transaction programs should be carefully developed and thoroughly tested
- During the running of the transaction
  - It is possible that the DB is in an *inconsistent* stateEx: consider money transfer between two bank accounts

# Isolation

- Let  $T_1, T_2, \dots, T_n$  be transactions submitted around the same time
- Isolation: Though the operations of  $T_i$  are interleaved with those of others, with respect to  $T_i$ ,
  - any other  $T_j$  *appears* to have either completed before  $T_i$  or started after  $T_i$  finished
- The operations of  $T_j$  are completely isolated from those of  $T_i$  and hence have no effect on  $T_i$
- Responsibility of Concurrency Control module

# Durability

Upon successful completion of a transaction,

System must ensure that its effect is permanently recorded in the database

Also effect of failed transactions is not recorded

Failures

- Transaction program failures
  - internal errors – attempted division by zero etc.
  - transaction aborts
- System crashes
  - power failures etc.

Responsibility of Recovery Module

# A Note on Transaction Sequencing

- Suppose  $T_1, T_2, \dots, T_n$  are transactions submitted around the same time
  - The transactions may not *end* in the same order
- From the system point of view, guaranteeing atomicity and isolation is important
- The ending time of each transaction depends on
  - data items being accessed
  - computation time etc.
  - system's policy for guaranteeing atomicity/ isolation
- From the submitter's point of view
  - when the transaction finishes matters!

# Concurrency Control Subsystem

Interleaving the operations of transactions

- essential to achieve good throughput

However, arbitrary interleaving of operations

- leads to inconsistent database

Certain regulated interleaving so that

- two transactions involving the use of same data item don't conflict with each other

Concurrency control subsystem ensures this

# Recovery Manager Subsystem

System Crash

- due to various reasons
- some transactions might have run partially

Transaction Failure

- transaction program error
- transaction was aborted by Concurrency Control Module due to violation of rules

Upon Recovery of system

- effect of partially-run transactions should not be there (atomicity)
- effect of completed transactions must be recorded (durability)

# Recovery Manager and System Log

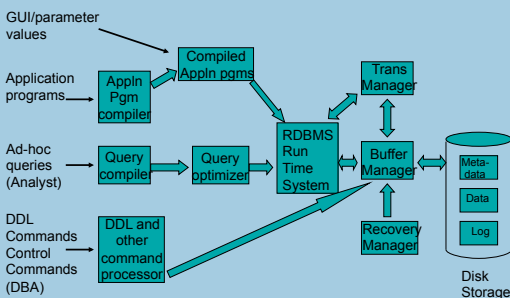
## System Log:

- Details of running transactions are recorded in the log
- Important resource for recovering from crashes
- Always maintained on reliable secondary storage

## Log Entries

- beginning of a transaction
- update operation details
  - old value, new value etc.
- ending of transaction
- more details later...

## Recall: Architecture of an RDBMS system



## Transaction Operations

From the DB system point of view, only the read and write operation of a transaction are important

- Other operations happen in memory and don't affect the database on the disk
- Notation:
  - R(X) - transaction Reads item X
  - W(X) - transaction Writes item X
- Database items – denoted by X, Y, Z, ...
  - Would be specified rigorously later

## Transaction Operation - *Commit*

- A transaction issues a *commit* command when
  - it has successfully completed its sequence of operations
  - indicates that its effect *can be* permanently recorded in the db
- Once the DBMS system allows a transaction to commit
  - System is obliged to ensure that the effect of the transaction is permanently recorded in the database
- What exactly happens during *commit*
  - depends on the specific error recovery method used and the specific concurrency control method used
  - will become clear later on in the module

## Transaction Operation - *Abort*

When a transaction issues *Abort*

- indicates that there is some internal error and
- transaction wants to terminate in the middle
  - For instance, a division by zero is being attempted by the txn
  - Or the debit account is going negative if we do money transfer etc

System obligation/ responsibility

- ensure that the partial work done by the transaction has no effect on the database on the disk

What exactly happens during *Abort*

- depends on the specific error-recovery method and the specific concurrency control method adopted by the system

## DB Model for Transaction Processing

- Database
  - Consists of several items – blocks / tuples
    - Granularity of item – affects the algorithms/protocols
    - Usually taken as a block/page
- Transactions operate by exchanging data with DB only
  - They do not exchange messages between them
- Focus on read/write/commit/abort operations
  - Ignore in-memory operations
- Transactions are not nested

## Transaction Operations

- $R_i(X)$  – read op of Txn  $i$ , reads db item  $X$ 
  - disk block having  $X$  - copied to buffer page - if reqd
  - required value is assigned to program variable  $X$
- $W_i(X)$  – write op of Txn  $i$ , writes db item  $X$ 
  - disk block having  $X$  - copied to buffer page - if reqd
  - update buffer value using program variable  $X$
  - transfer block to disk – immediately or later
- $C_i$  – commit of Txn  $i$
- $A_i$  – abort of Txn  $i$

## Need for Concurrency Control

- If operations of transactions are interleaved arbitrarily
  - Several problems / anomalies arise
  - Can be classified as
    - WR anomalies
    - RW anomalies
    - WW anomalies

## WR Anomalies or Dirty Reads

- Txn  $T_1$  is in progress; updating values in a column
- Txn  $T_2$  reads a value  $X$  updated by  $T_1$ , uses it to compute some other quantity, and finishes!
- $T_1$  for some reason changes  $X$  back to its original value!
  - $T_2$  has read *dirty* data or intermediate data

## RW Anomalies or Unrepeatable Reads


- $T_1$  has read a value  $X$  and intends to read it again before changing it
- Between two reads of  $X$  by  $T_1$ ,  
 $T_2$  reads  $X$ , modifies it and finishes
- $T_1$  reads  $X$  the 2<sup>nd</sup> time and gets a different value!
  - *Unrepeatable read* problem
  - $X$  could be the number of seats available for reservation

## WW Anomalies or Lost Updates

- $T_1$  reads an item  $X$ , reduces it by 10% and wants to write it
- $T_2$  reads the same  $X$  before it was updated by  $T_1$ , increments it by 20% and wants to write it
- Say *write* of  $T_1$  happens and then that of  $T_2$ 
  - Final value of  $X$  is  $1.2 * X$ !
  - $T_1$ 's Update of  $X$  is lost – *lost update* problem

## Transaction Schedules

- Sequence of interleaved operations of a Txn set
    - Ops of a Txn  $T$  appear in the same order as in  $T$
- $R_1(X) R_2(Y) R_3(X) W_1(X) W_3(X) W_2(Y)$



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

- Serial :  $R_1(X) W_1(X) R_2(Y) W_2(Y) R_3(X) W_3(X)$ 
  - No interleaving



# Serializability

- Serial schedules
  - No interleaving of operations of different txns
  - Do not cause any concurrency problems
  - But performance (throughput) is low
- Serializable Schedules
  - Interleaving of operations happens
  - But, in *some* sense *equivalent* to serial schedules
    - The effect of the interleaving is same as that of *some* serial schedule

## Conflicting Pairs of Operations

- In a schedule, a pair of operations are said to be in *conflict* if
  - The operations belong to two different txns
  - Both the operations deal with the same DB item
  - One of the two ops is a Write operation

- 1) R(X) of T1 conflicts with W(X) of T3
- 2) R(X) of T3 conflicts with W(X) of T1
- 3) W(X) of T1 conflicts with W(X) of T3

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

## Conflict Equivalence

A schedule  $S_1$  is said to *conflict-equivalent* to  $S_2$  if  
The relative order of *any* conflicting pair of operations is the *same* in both  $S_1$  and  $S_2$

$R_1(X) R_2(Y) \underline{R_3(X)} W_1(X) \underline{W_2(X)} W_2(Y)$  (in the table)  
is conflict equivalent to

$R_1(X) R_2(Y) \underline{R_3(X)} W_2(Y) W_1(X) \underline{W_2(X)}$

But is not conflict-equivalent to

$R_1(X) R_2(Y) W_1(X) \underline{R_3(X)} \underline{W_2(X)} W_2(Y)$   
(violet pair is out of order)

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

## Conflict Serializable Schedules

- A schedule is called *conflict serializable* if it is conflict equivalent to *some* serial schedule
- Or, if we can move non-conflicting ops such that
  - The relative order of ops of a Txn is intact and
  - Schedule becomes a serial
  - Schedule in picture
    - is *not* conflict-serializable
    - T1 < T3 disturbs the 2<sup>nd</sup> pair and
    - T3 < T1 disturbs the 1<sup>st</sup> and 3<sup>rd</sup> pairs

T1	T2	T3
R(X)	1	
	R(Y)	
	2	R(X)
W(X)	3	W(X)
	W(Y)	

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

28

## Conflict Serializable Schedules

Suppose we make a slight change to our example:

S1	T1	T2	T3
	R(X)		
		R(Y)	
			R(X)
	W(X)		
			W(X)
		W(Y)	

→

S2	T1	T2	T3
	R(X)		
		R(Y)	
	W(X)		
			R(X)
			W(X)
		W(Y)	

S2 is conflict-equivalent to serial schedules

T1,T3,T2 ; T1,T2,T3 and T2,T1,T3

Non-conflicting pairs of ops -- moved to get these serial schedules

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

29

## Precedence Graph of a Schedule

- Precedence graph or serialization graph
- Nodes represent transactions
- A directed arc exists from  $T_i$  to  $T_j$  if
  - An operation of  $T_i$  *precedes* an operation of  $T_j$  and *conflicts* with it
- A schedule S is conflict-serializable if and only if the precedence graph of S is *acyclic*
  - Topological sorts of the graph give equivalent serial schedules

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

30

## Precedence Graph - Examples

Precedence graph (or serialization graph) examples

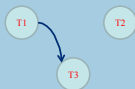
S1

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



S2

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

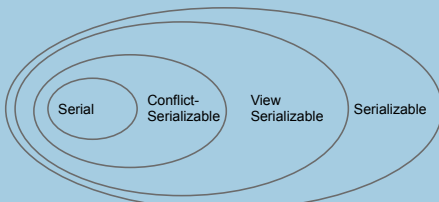


Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

31

## Conflict Serializability and Serializability

- Conflict serializability is a sufficient condition for serializability but is not necessary!



Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

32

## View Serializability

- A schedule  $S_1$  is view-equivalent to  $S_2$  if any
  - $T_i$  read the initial value of a db item  $X$  in  $S_1$ , it does so in  $S_2$  also.
  - $T_i$  read a db item  $X$  written by  $T_j$  in  $S_1$ , it does so in  $S_2$  also.
 and,
  - For each db item  $X$ , the txn that wrote the final value of  $X$  is same in  $S_1$  and in  $S_2$ .
- A schedule is view-serializable if it is view-equivalent to *some* serial schedule

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

33

# Concurrency Control Using Locks

- Assumptions
  - A transaction requests for a lock on a db item X before doing either *read* or *write* on X
  - A transaction unlocks X after it is done with X
- Locks - binary locks are assumed
  - Ensure mutual exclusion
  - At any time, at most one transaction holds a lock on a db item
    - Locking scheduler – ensures above and
    - Keeps track of who holds lock on what item

## Locking and Serializability

T1	T2	T3
L(X),R(X),U(X)		
	L(Y),R(Y),U(Y)	
		L(X),R(X),U(X)
L(X),W(X),U(X)		
		L(X),W(X),U(X)
	L(Y),W(Y),U(Y)	

- Locking alone
  - Does not guarantee serializability
  - Above schedule continues to be non-serializable

## Two Phase Locking (2PL)

- 2PL Protocol
  - All *lock* requests of a transaction precede the first *unlock* request
  - Or a transaction has a *locking* phase followed by an *unlocking* phase
- If all transactions follow 2PL protocol
  - The *resulting* schedules will be conflict-serializable
- A very important and valuable result!

## Why 2PL works?

- $S$  : A schedule of  $n$  transactions that follow 2PL
- Let  $T_i$  be the transaction
  - that issues the first *unlock* request among all txns
- We will argue that
  - All ops of  $T_i$  can be brought to the beginning of  $S$  without passing over any conflicting ops
- We get  $S_1$ : (ops of  $T_i$ ); (ops of other  $n-1$  txns)
  - $S_1$  is conflict-equivalent to  $S$
- Thus we can show that  $S$  is conflict-serializable

## Why 2PL works?

- Suppose some op of  $T_i$ , say  $W_i(X)$ , is conflicting with some *preceding* op, say  $W_j(X)$ , of  $T_j$  in  $S$   
...  $W_j(X)$ , ...,  $W_i(X)$ ,... or we have,  
...  $W_j(X)$ , ...,  $U_j(X)$ , ...,  $L_i(X)$ , ...,  $W_i(X)$ ,...
- As  $T_i$  is the *first* txn to issue an unlock, say,  $U_i(Y)$ ,
  - $U_i(Y)$  precedes  $U_j(X)$  in  $S$
- So,  $S$  could be  
...  $W_j(X)$ , ...,  $U_i(Y)$ , ...,  $U_j(X)$ , ...,  $L_i(X)$ , ...,  $W_i(X)$ ,...
- Then,  $T_i$  is not following 2PL – a contradiction

## Why 2PL works?

- The argument is same for any conflicting pairs of operations involving an operation of  $T_i$
- Thus, for all ops of  $T_i$ , there are no conflicting ops that precede it.
- So, beginning with the first op of  $T_i$ ,
  - We can swap ops of  $T_i$  with the previous ops and bring them all the way to the front of schedule  $S$
- This can be repeated among the remaining  $n-1$  txs
- $S$  is conflict-equivalent to a serial schedule

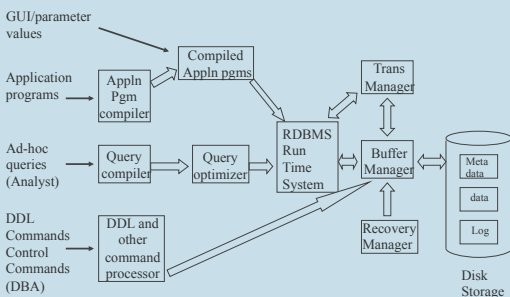
## Possibility of Deadlocks

- Use of basic 2PL
  - Deadlocks may occur; Deadlock detection and resolution is adopted
- To detect
  - A graph called *wait-for* graph is maintained
  - Each running Txn – a node
  - If  $T_i$  is waiting to lock an item held by  $T_j$ 
    - Directed edge from  $T_i$  to  $T_j$
- To resolve
  - select Txn in cycles & abort/resubmit them

## System Failures and Recovery

- Failures
  - Transaction errors – abrupt ending w/o completing
  - CC module may decide to abort a Txn
  - Disk crashes/power failures
- Log – the principal tool for error recovery
  - Sequential file
    - log entries reach the log on disk in same order as they are written (an important assumption)
  - Undo Logs / Redo Logs / Undo-Redo logs
  - Each with a corresponding recovery method

## Architecture of an RDBMS system



## Buffer Manager

- DB item – a disk block
- Disk blocks – brought into Memory Buffers
  - Modified by running transactions
  - Written to disk when transaction completes
- We will use more detailed Txn operations
  - Input(A): get disk block with A into buffer
  - Read(A, t):  $t := A$ ; do Input(A) if reqd; t is local var
  - Write(A, t):  $A := t$ ; do Input(A) if reqd
  - Output(A): send block with A from buffer to disk

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

## Log Record Entries

- <Start T> -- Txn T has begun
- <End T> -- Txn T has ended
- <Commit T>
  - T has successfully completed its work
  - Changes made by T must appear in the on-disk DB
- <Abort T>
  - T has not successfully completed its work
  - Changes done by T shouldn't be there in the on-disk DB
- Update records – *specific* to the logging method
- Flush Log – Force-write log entries to the disk

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

44

## Undo Logging

### Update Type Log Record

<T, X, V> : Txn T has changed the item X and  
its *old* value is V

### Undo Logging Rules:

- $U_1$ : If a transaction T modifies db item X, then the log record <T, X, V> must be written to disk **before** the new value of X is written to disk.
- $U_2$ : If a transaction T commits, then its **COMMIT** log record must be written to disk only **after** all db items changed by T have been written to disk, but as soon thereafter as possible.

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

45

## Undo Logging

When a Txn T Commits,  
DB items flow to disk as below:

for each modified DB item X  
    { send update entry  $\langle T, X, V \rangle$  to disk  
      write item X to disk }  
write  $\langle \text{Commit } T \rangle$  to disk

## Undo Logging Example

Txn T is doing money transfer of Rs 100/- from account A to account B  
Consistency Requirement:  $A+B$  is same before and after the Txn

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							$\langle \text{start } T \rangle$
2	Read(A,m)	500	500		500	1500	
3	$m := m - 100$	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	$\langle T, A, 500 \rangle$
5	Read(B,m)	1500	400	1500	500	1500	
6	$m := m + 100$	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	$\langle T, B, 1500 \rangle$
8	Flush Log						
9	Output(A)	1600	400	1600	400	1500	
10	Output(B)	1600	400	1600	400	1600	
11							$\langle \text{commit}, T \rangle$
12	Flush Log						

## Recovery Using Undo Log

Examine Log and Partition Txns into:

Committed Set : Txns for which  $\langle \text{Commit } T \rangle$  exists

Incomplete Set : Txns for which  $\langle \text{Abort } T \rangle$  exists or  
 $\langle \text{Commit } T \rangle$  does not exist

Examine Log in the *reverse* direction

For every update record  $\langle T, X, V \rangle$

T is Committed: **Do nothing**; All is well due to  $U_2$  !

T is Incomplete: Restore value of X on disk as V

- T might have changed some items on disk

- But log entries with old values are on disk due to  $U_1$

Do  $\langle \text{Abort } T \rangle$  log entry for each incomplete T & Flush Log



## Undo Logging: Crash Recovery

Crash occurs sometime before the first Flush Log:

DB on disk – unchanged; T - recognized as incomplete;

Log entries of T - unsure if they are on disk; if present - used for “undo”;  
no harm! <Abort T> entered; T – resubmitted;

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:=m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,500>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1500>
8	Flush Log						
9	Output(A)	1600	400	1600	400	1500	
10	Output(B)	1600	400	1600	400	1600	
11							<commit,T>
12	Flush Log						

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

49

## Undo Logging: Crash Recovery

Crash occurs sometime after the first Flush-Log but before Step 11:

DB on disk - might have changed; T - recognized as incomplete;

Log entries of T - on disk; used for undoing T;

<Abort T> entered; T resubmitted;

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:=m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,500>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1500>
8	Flush Log						
9	Output(A)	1600	400	1600	400	1500	
10	Output(B)	1600	400	1600	400	1600	
11							<commit,T>
12	Flush Log						

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

50

## Undo Logging: Crash Recovery

Crash occurs after Step 11, before Step 12: DB on disk - changed;

<Commit T> - on disk: T - recognized as completed; No action reqd;

<Commit T> - not on disk: T - recognized as incomplete; Log entries of  
T used for undoing T; <abort T> entered; T resubmitted;

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:=m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,500>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1500>
8	Flush Log						
9	Output(A)	1600	400	1600	400	1500	
10	Output(B)	1600	400	1600	400	1600	
11							<commit,T>
12	Flush Log						

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

51

# Undo Logging: Crash Recovery

Crash occurs after Step 12:

DB on disk - changed;

<Commit T> - on disk: T - recognized as completed; No action reqd;

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:=m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,500>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1500>
8	Flush Log						
9	Output(A)	1600	400	1600	400	1500	
10	Output(B)	1600	400	1600	400	1600	
11							<commit,T>
12	Flush Log						

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

52

## Redo Logging

### Undo Logging:

Cancels the effect of incomplete Txns and

Ignores the committed Txns

All DB items to be sent to disk before Txn can commit

Results in lot of I/O; called FORCE-writing;

### Redo Logging:

Ignores incomplete Txns and

Repeats the work of Committed Txns

Update log entry <T, X, V> : V is the **new** value

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

53

## Redo Logging

### Redo Logging Rule:

Before modifying any database element X on disk,  
it is necessary that all log records pertaining to  
this modification of X, including both the update  
record <T, X, V> and the <COMMIT T> record,  
**must** appear on disk.

Also called the **write-ahead logging** (WAL) rule

Items flow like this: Update Log records,  
the commit entry  
and then the changed DB items.

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

54

## Redo Logging Example

Txn T is doing money transfer of Rs 100/- from account A to account B  
Consistency Requirement: A+B is same before and after the Txn

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:=m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,400>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1600>
8							<commit,T>
9	Flush Log						
10	Output(A)	1600	400	1600	400	1500	
11	Output(B)	1600	400	1600	400	1600	

## Recovery Using Redo Log

Examine Log and Partition Txns into:

Committed Set : Txns for which <Commit T> exists

Incomplete Set : Txns for which <Abort T> exists or  
<Commit T> does not exist

Examine Log in the *forward* direction

For every update record <T,X,V>

T is Incomplete: **Do nothing**; DB on disk has no effects!

T is Committed: Unsure if all the effects of T are on disk

- But log entries with new values are on disk (WAL)
- Redo the change as per the log entry

Do <Abort T> log entry for each incomplete T & Flush Log

## Redo Logging: Crash Recovery

Crash Occurs Before Step 8: <Commit T> not made by T;  
T is recognized as incomplete; No action reqd; DB on disk - not  
changed (WAL); enter <Abort T>; resubmit T

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:=m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,400>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1600>
8							<commit,T>
9	Flush Log						
10	Output(A)	1600	400	1600	400	1500	
11	Output(B)	1600	400	1600	400	1600	

## Redo Logging: Crash Recovery

Crash Occurs after Step 8:

<Commit T> - on disk; T is redone with the help of Log entries;

<Commit T> not on disk: T is treated as incomplete; No action reqd;

DB on disk - not changed (WAL); enter <abort T>; resubmit T

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:=m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,400>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1600>
8							<commit,T>
9	Flush Log						
10	Output(A)	1600	400	1600	400	1500	
11	Output(B)	1600	400	1600	400	1600	

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

58

## Redo Logging: Crash Recovery

Crash Occurs after Step 9:

<Commit T> - surely on disk; T is redone with the help of Log entries;

Whether or not T succeeded in writing them!

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:=m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,400>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1600>
8							<commit,T>
9	Flush Log						
10	Output(A)	1600	400	1600	400	1500	
11	Output(B)	1600	400	1600	400	1600	

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

59

## Undo-Redo Logging

### Undo Logging:

Cancels the effect of incomplete Txns

All DB items to be sent to disk before Txn can commit

Results in a lot of I/O -- I/O can not be “bunched”

### Redo Logging:

Ignores incomplete Txns and

Repeats the work of Committed Txns

Buffer might contain the blocks of several committed

Txns – Buffer utilization might come down!

### Undo-Redo Logging:

Better flexibility; uses more detailed logging

Prof P Sreenivasa Kumar,  
Department of CS&E, IITM.

60

# Undo-Redo Logging

Undo-Redo Logging Update Entry:

$\langle T, X, U, V \rangle$  :

Txn T has changed the db item X and  
its old value is U and new value is V

UR Logging Rule:

Before modifying any database element X on disk, because of changes made by some transaction T, it is necessary that the update record  $\langle T, X, U, V \rangle$  appears on disk.

$\langle \text{Commit } T \rangle$  and disk changes – in any order!

# Undo-Redo Logging Example

Txn T is doing money transfer of Rs 100/- from account A to account B  
Consistency Requirement:  $A+B$  is same before and after the Txn

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							$\langle \text{start } T \rangle$
2	Read(A,m)	500	500		500	1500	
3	$m := m - 100$	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	$\langle T, A, 500, 400 \rangle$
5	Read(B,m)	1500	400	1500	500	1500	
6	$m := m + 100$	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	$\langle T, B, 1500, 1600 \rangle$
8	Flush Log						
9	Output(A)	1600	400	1600	400	1500	
10							$\langle \text{commit}, T \rangle$
11	Output(B)	1600	400	1600	400	1600	

# Recovery Using Undo-Redo Logs

Examine Log and Partition Txns into:

Committed Set : Txns for which  $\langle \text{Commit } T \rangle$  exists

Incomplete Set : Txns for which  $\langle \text{Abort } T \rangle$  exists or  
 $\langle \text{Commit } T \rangle$  does not exist

Recovery Method:

Redo all committed Txns – order – earliest first

Undo all incomplete Txns – order – latest first

Necessary to do both!

# Undo-Redo Log: Crash Recovery

Crash before <Commit T> is on disk: Txn T – incomplete – undone.  
Crash after <Commit T> is on disk: Txn T – completed – redone.

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:=m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,500,400>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1500,1600>
8	Flush Log						
9	Output(A)	1600	400	1600	400	1500	
10							<commit,T>
11	Output(B)	1600	400	1600	400	1600	

## Issue of Dirty Data and Commits

- Consider Txns T and S:
  - T has modified a db item X and it is doing some more work
  - Meanwhile, S has read X, completed its work
  - Suppose S is allowed to Commit
  - Now, T has an internal error and decides to Abort
  - S has read ‘dirty’ data
  - But S can’t be *undone* as it was allowed to *commit*
- DB got into a trouble!

## Recoverable Schedules

- A schedule S is called **recoverable** if no Txn T in S *commits* until all the Txns, that have written an item T reads, have committed
  - T needs to wait till each of the Txn from which it has read completes.
  - If all commit, then T can go ahead and commit
  - If at least one such Txn aborts, T also has to abort
  - Cascading Aborts/Rollbacks may occur
- Recoverability is an essential requirement!!

## Recoverability vs Serializability

- Orthogonal concepts
- Both are important for a Transaction System!
- It is possible that a recoverable schedule is not conflict-serializable
  - Recoverability defn has no restrictions on locking
- It is also possible that a serializable schedule is not recoverable
  - Serializability definition has no restriction on committing

## Cascadeless Schedules

- A recoverable schedule
  - may result in cascading rollbacks or aborts
- A schedule is called *cascadeless* or ACR (avoiding cascading rollbacks) if
  - in the schedule, Txns read only values written by *committed* Txns
- Every ACR schedule is recoverable
  - Such a Txn surely commits only after all the Txns it has read from commit; in fact it does not read values written by uncommitted Txns.

## Strict Schedules

- A schedule is called *strict* if in the schedule
  - a Txn neither reads nor writes an item X until the last Txn that writes X has terminated
- Strict 2PL:
  - A Txn must not release any write locks until the Txn has either committed or aborted and the commit or abort log record has been written to disk
  - Results in strict schedules
  - Strict schedules are *cascadeless* and *serializable*

## Transactions in SQL

- Important parameters of Transactions in SQL:
  - Can be set for each transaction
  - Access-Mode:
    - Read-Only; Read-Write (default)
  - Isolation Level: Default is *serializable*
    - Other lower isolation levels are also available
    - Meant for running transactions that collect statistics
    - For isolation level “uncommitted” – Read-Only Txns only
- Each transaction ends with Commit or Rollback

## Transaction isolation levels

Level	Dirty Reads	Unrepeatable Reads	Phantoms
Read Uncommitted	May Be	May Be	May Be
Read Committed	No	May Be	May Be
Repeatable Read	No	No	May Be
Serializable	No	No	No

## Phantom Records

- Phantom records problem:
  - Txn T1 has selected a set of tuples based on a certain condition C, say “student.Dept = 5”
  - And is working with them, say get max(marks)
  - Txn T2 updated the DB with a new tuple that satisfies C after T1 started
  - Can cause T1 to be incorrect
  - Such rows are called *phantom* rows
    - Come into picture out of the blue...
  - “index locking” needs to be adopted