CSC553 Advanced Database Concepts

Tanu Malik

School of Computing

DePaul University

Serial Schedules

- So far we have seen sufficient conditions that allows us to check whether the schedule is serializable.
  - Serial
  - Serializable
  - Conflict serializable
  - View serializable
  - Recoverability
    - Recoverable
    - Avoids cascading deletes

Scheduler

- The scheduler:
  - Module that schedules the transaction’s actions, ensuring serializability

Scheduler needs CC

- Two main approaches
  - Pessimistic CC:
    - Lock-based concurrency control needs deadlock detection
      - Prevents unserializable schedules
      - Never abort for serializability (but may abort for deadlocks)
      - Best for workloads with high levels of contention
  - Optimistic:
    - Timestamp-based concurrency control
    - Tracking of read-set/write-set, validation before commit.
      - Assume schedule will be serializable
      - Abort when conflicts detected
      - Best for workloads with low levels of contention

  - Multi-version: less concurrency overhead for read-only queries

Timestamp-based CC—High level

1. Assign a timestamp to each transaction (txn).
2. Record the timestamps of the txn that last read or wrote a database object O.
3. Ensure that actual schedule (wrt read/write elements) is equivalent to a serial schedule according to txn timestamps.
   - Otherwise rollback.

The timestamp order of elements defines the serialization order of the transactions
Will generate a schedule that is view-equivalent to a serial schedule, and recoverable

Timestamps

- Each transaction receives unique timestamp TS(T)
  - When it enters the system
- What does timestamp tell us:

  A unique order

- Could be:
  - The system's clock
  - A unique counter, incremented by the scheduler

T_1: 10:00, T_2: 10:01,
T_3: 10:04

TS(T_1): 100 Oldest
TS(T_1): 200
TS(T_3): 300
t Youngest

Timestamps

- With each database object X, associate
- RT(X) = the highest timestamp of any transaction U that read X
  - The timestamp of the last (most recent) transaction which performed read successfully
- WT(X) = the highest timestamp of any transaction U that wrote X
  - The timestamp of the last (most recent) transaction which performed read successfully

- **If transactions abort, we must reset the individual timestamps**

Example

- $TS(T_1)$: 10; $TS(T_2)$: 20; $TS(T_3)$: 30

T_1	T_2	T_3
R(A)		
	R(A)	
		R(A)

RT(A) = 30

Example

- $TS(T_1)$: 10; $TS(T_2)$: 20; $TS(T_3)$: 30

T_1	T_2	T_3
W(A)		
		W(A)
	W(A)	

WT(A) = ?

Example

- $TS(T_1)$: 10; $TS(T_2)$: 20; $TS(T_3)$: 30

T_1	T_2	T_3
W(A)		
		W(A)
	W(A)	
	commit	

WT(A) = ?

C(X) = ?

Timestamp-based CC—High level

1. Assign a timestamp to each transaction (txn).
2. Record the timestamps of the txn that last read or wrote a database object O.
3. Ensure that actual schedule (wrt read/write elements) is equivalent to a serial schedule according to txn timestamps.
   - Otherwise, rollback.

Conflicts acceptable according to Txn timestamps

- The transaction that comes earlier must also complete earlier according to read/write timestamps.
  - Older txns are given priority.

$T_1 = 100$	$T_2 = 200$
$r_1(A)$	
	$w_2(A)$

$T_1 = 100$	$T_2 = 200$
$w_1(A)$	
	$r_2(A)$

$T_1 = 100$	$T_2 = 200$
$w_1(A)$	
	$w_2(A)$

Let these proceed under the assumption that they will commit

Conflicts in Timestamps

- For any $r_T(X)$ or $w_T(X)$ request, check for conflicts:

- $w_U(X)$ . . . $r_T(X)$
- $r_U(X)$ . . . $w_T(X)$
- $w_U(X)$ . . . $w_T(X)$

How to check if read is too late?

Or write is too late?

Conflicts in Timestamps

- For any $r_T(X)$ or $w_T(X)$ request, check for conflicts:

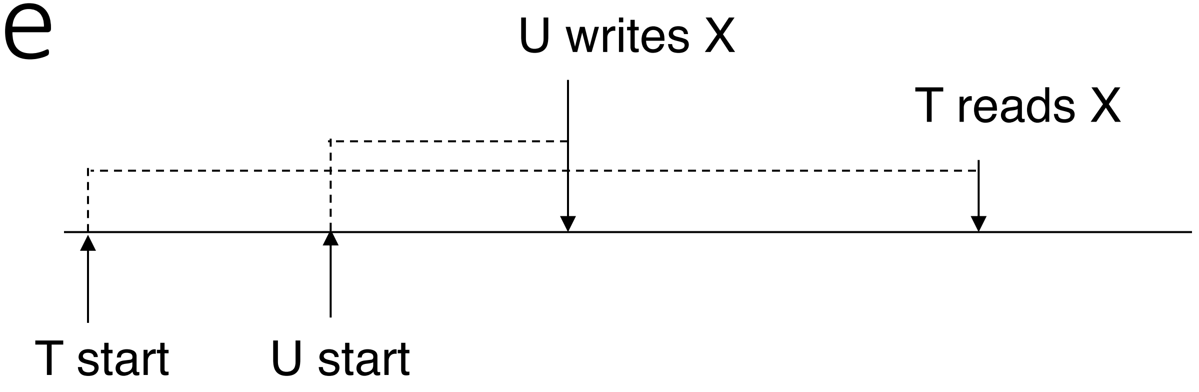
- $w_U(X) \ldots r_T(X)$
- $r_U(X) \ldots w_T(X)$
- $w_U(X) \ldots w_T(X)$

How to check if read is too late?
Or write is too late

Fundamental Observation:

When T requests $r_T(X)$ or $w_T(X)$, need to check $TS(U) \leq TS(T)$

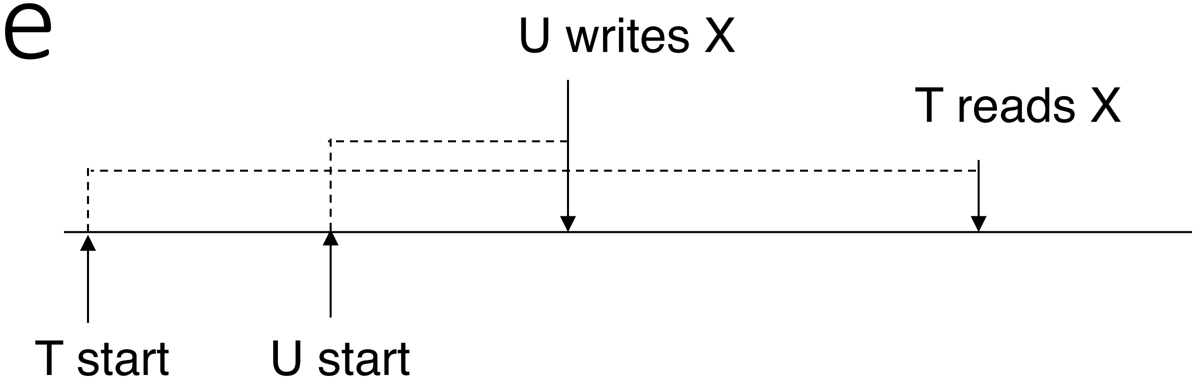
Example: Read too late



- T wants to read X

- begin(T)… begin(U) … w_U(X)… r_T(X)

Example: Read too late



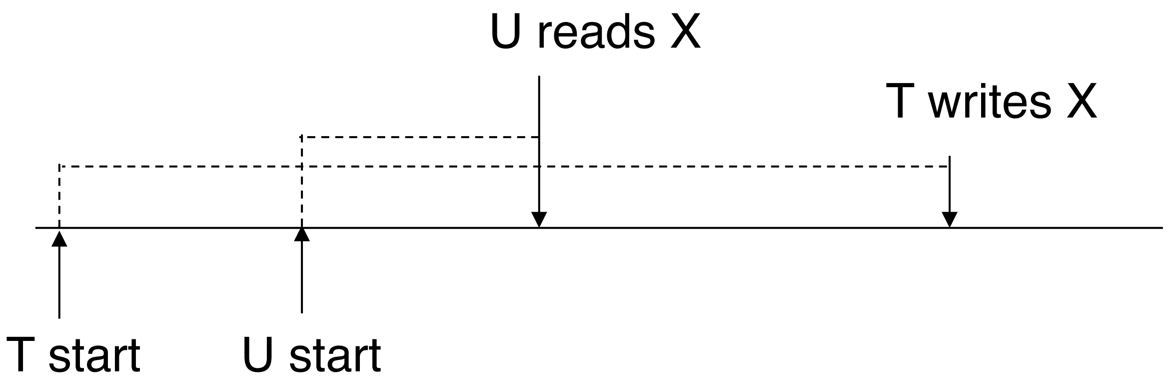
T should not be allowed to perform a non-serializable read
Therefore, rollback T

- T wants to read X
- begin(T)… begin(U) … $w_U(X)$… $r_T(X)$
- If WT(X) (= TS(U)) > TS(T) then need to rollback T !
- T tried to read too late!

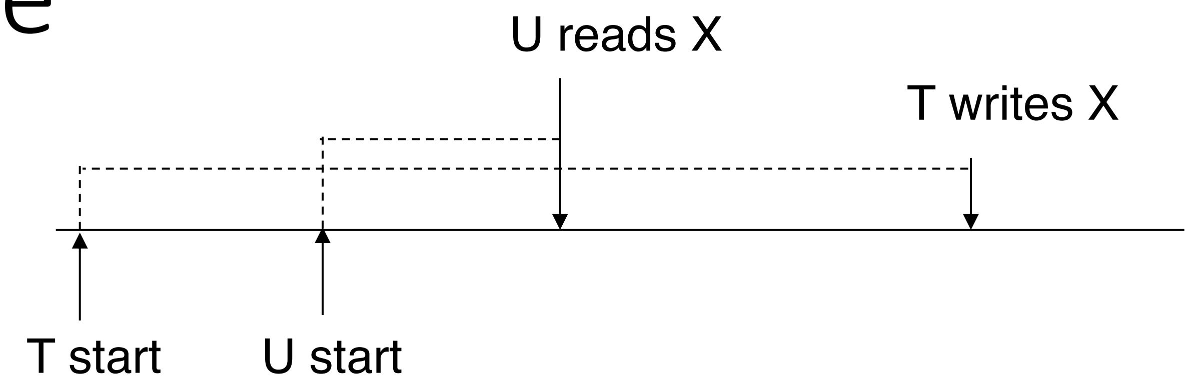
Example: Write too late

- T wants to write X

- begin(T)… begin(U) … $r_U(X)$… $w_T(X)$



Example: Write too late



- T wants to read X

U should not be allowed to perform an inconsistent read
Therefore, rollback T

- begin(T)… begin(U) … $r_U(X)$… $w_T(X)$

- If RT(X) (= TS(U)) > TS(T) then need to rollback T.
- T tried to write too late!

Conflict Serializability

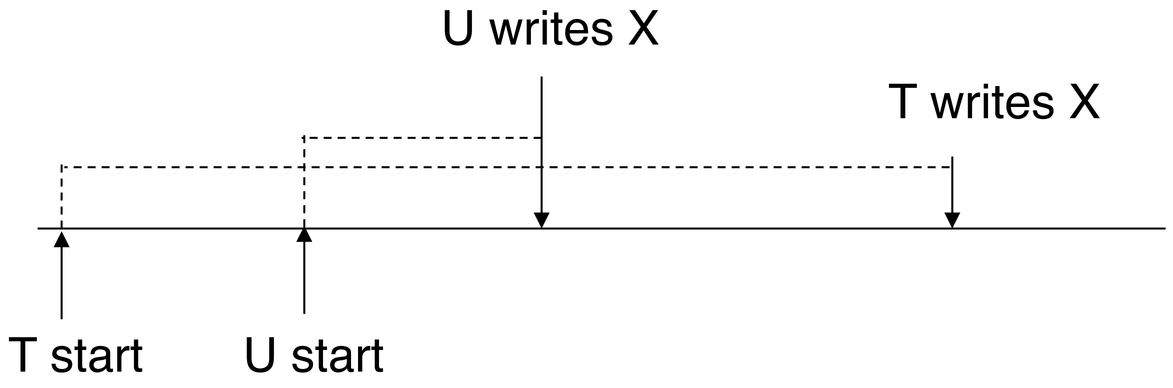
- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.

Thomas Rule for Write-Write Conflict

- But… we do not need to rollback in one case:
- T wants to write X
- START(T) … START(U) … $w_U(X)$ . . . $w_T(X)$



Thomas Rule

- But... we can still handle it in one case:
- T wants to write X
- START(T) … START(U) … $w_U(X)$ . . . $w_T(X)$

WT(X) (= TS(U)) **> TS(T) then don't write X at all!**

Thomas Rule

- But… we can still handle it in one case:
- T wants to write X
- START(T) … START(V)…START(U) … $W_U(X)$ . . .$R_V(X)$… $W_T(X)$

Because no other transaction V that should have read T's value got U's value since V would have been aborted because of too-late read.

If RT(X) ≤ TS(T) and WT(X) (= TS(U)) **> TS(T) then don't write X at all!**

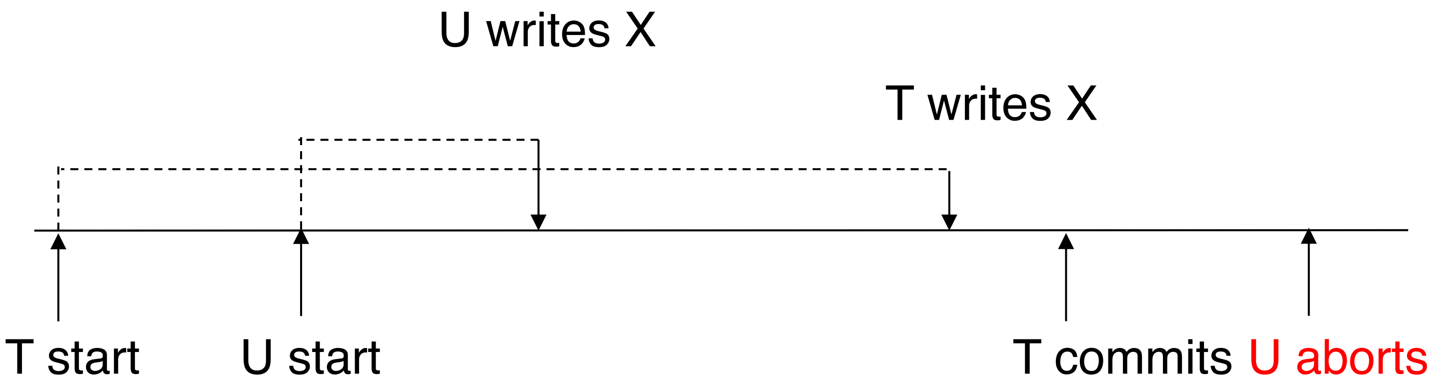
Summary so far

- Only for transactions that do not abort
  - Otherwise, may result in non-recoverable schedule
- Transaction wants to READ element X
  - If WT(X) > TS(T) then ROLLBACK
  - Else READ and update RT(X) to larger of TS(T) or RT(X)
- Transaction wants to WRITE element X
  - If RT(X) > TS(T) then ROLLBACK
  - Else if WT(X) > TS(T) ignore write & continue (Thomas Write Rule)
  - Else, WRITE and update WT(X) =TS(T)

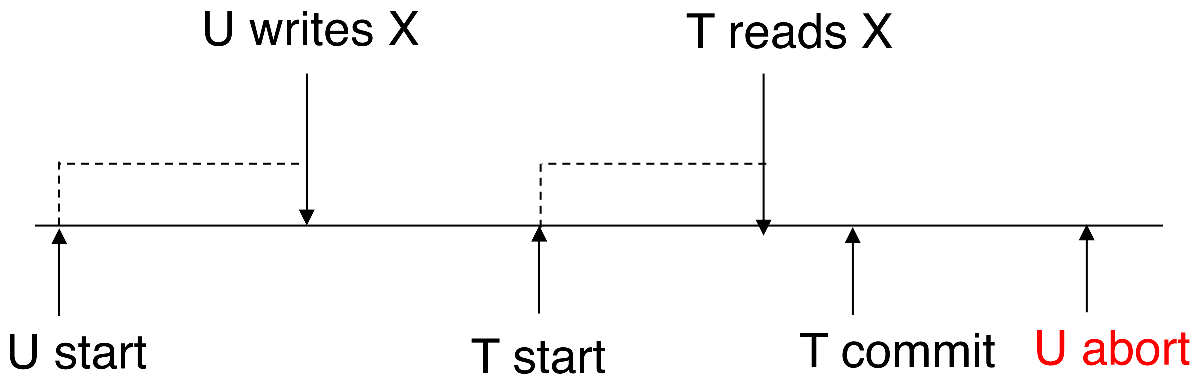
How to deal with aborts?

- C(X) = the commit bit: true when transaction with highest timestamp that **wrote** X committed
  - True if the last (most recent) transaction committed.

Dealing with Aborts-Case 1 (Lost update)



Dealing with Aborts-Case 2: Dirty Reads



Ensuring Recoverable Schedules

- Use the commit bit C(X) to keep track if the transaction that last wrote X has committed (just a read will not change the commit bit)

- Recall:

- Schedule avoids cascading aborts if whenever a transaction reads an element, then the transaction that wrote it must have already committed

Ensuring Recoverable Schedules

- Read dirty data:

- T wants to read X, and WT(X) < TS(T)
- Seems OK, but…

- START(U) … START(T) … W_U(X). . . R_T(X)… ABORT(U)

- If C(X)=false, T needs to wait to commit for it to become true

Ensuring Recoverable Schedules

- Thomas' rule needs to be revised:
- T wants to write X, and WT(X) > TS(T)
- Seems OK not to write at all, but …

- START(T) … START(U)… W_U(X). . . W_T(X)… ABORT(U)

- If C(X)=false, T needs to wait for it to become true

Timestamp-based Scheduling

- When a transaction T requests $R_T(X)$ or $W_T(X)$, the scheduler examines RT(X), WT(X), C(X), and decides one of:

- To grant the request, or
- To rollback T (and restart with later timestamp)
- To delay T until C(X) = true

Timestamp-based Scheduling

Transaction wants to READ element X

- If WT(X) > TS(T) then ROLLBACK
- Else If C(X) = false, then WAIT
- Else READ and update RT(X) to larger of TS(T) or RT(X)

Transaction wants to WRITE element X

- If RT(X) > TS(T) then ROLLBACK
- Else if WT(X) > TS(T) Then
  - If C(X) = false then WAIT
  - Else IGNORE write (Thomas Write Rule)
- Otherwise, WRITE, and update WT(X)=TS(T), C(X)=false

Example

T1	T2	T3	T4	A
150	200	175	225	RT= 0, WT= 0
R(A)				RT = 150
W(A)				WT = 150
	R(A)			RT = 200
	W(A)			WT = 200
		R(A)		
		Abort		
			R(A)	RT = 225

WT > TS(T3)

T1	T2	T3	T4	
1	2	3	4	RT(A) = 0 WT(A) = 0 C = true

T1	T2	T3	T4	
1	2	3	4	RT(A) = 0 WT(A) = 0 C = true
	W(A)			
R(A)				
		R(A)		
	commit			
		R(A)		
			W(A)	
		W(A)		
			abort	
		W(A)		

T1	T2	T3	T4	
1	2	3	4	RT(A) = 0 WT(A) = 0 C = true
	W(A)			RT(A) = 0 WT(A) = 2 C = false
R(A)				
abort				
		R(A) **delay**		
	commit			RT(A) = 0 WT(A) = 2 C = true
		R(A)		RT(A) = 3 WT(A) = 2 C = true
			W(A)	RT(A) = 0 WT(A) = 4 C = false
		W(A) **delay**		
			abort	RT(A) = 0 WT(A) = 2 C = true
		W(A)		RT(A) = 0 WT(A) = 3 C = false

Summary of Timestamp-based Scheduling

- View-serializable
- Avoids cascading aborts (hence: recoverable)
- Does NOT handle phantoms
  - These need to be handled separately, e.g. predicate locks

Multiversion Timestamp

- When transaction T requests R(X)
- but WT(X) > TS(T), then T must rollback

- Idea: keep multiple versions of X: X_t , X_{t-1}, X_{t-2}, . . .

- $TS(X_t) > TS(X_{t-1}) > TS(X_{t-2}) > \ldots$

Details

- When $w_T(X)$ occurs, if the write is legal then create a new version, denoted X_t where t = TS(T)

- When $r_T(X)$ occurs, find most recent version X_t such that t <= TS(T)
  - $WT(X_t)$ = t and it never changes for that version
  - $RT(X_t)$ must still be maintained to check legality of writes

Example

T1	T2	T3	T4	A
150	200	175	225	RT= 0, WT= 0
R(A)				RT = 150
W(A)				WT = 150
	R(A)			RT = 200
	W(A)			WT = 200
		R(A)		
		Abort		
			R(A)	RT = 225

WT > TS(T3)

Example

T1	T2	T3	T4	A
150	200	175	225	RT= 0, WT= 0
R(A)				RT = 150
W(A)				WT = 150; Create X1
	R(A)			RT = 200
	W(A)			WT = 200; Create X2
		R(A) ←X1		RT = 200
		W(A); Abort		
			R(A)	RT = 225

Example

T1	T2	T3	T4	A
150	200	175	225	RT= 0, WT= 0
R(A)				RT = 150
W(A)				WT = 150
	R(A)			RT = 200
	W(A)			WT = 200
		R(A)		
		Abort		
			R(A)	RT = 225

Transaction Management

- Two parts:
  - Concurrency control: ACID
  - Recovery from crashes: ACID
- We already discussed concurrency control You are implementing locking in lab3
- Today, we start recovery

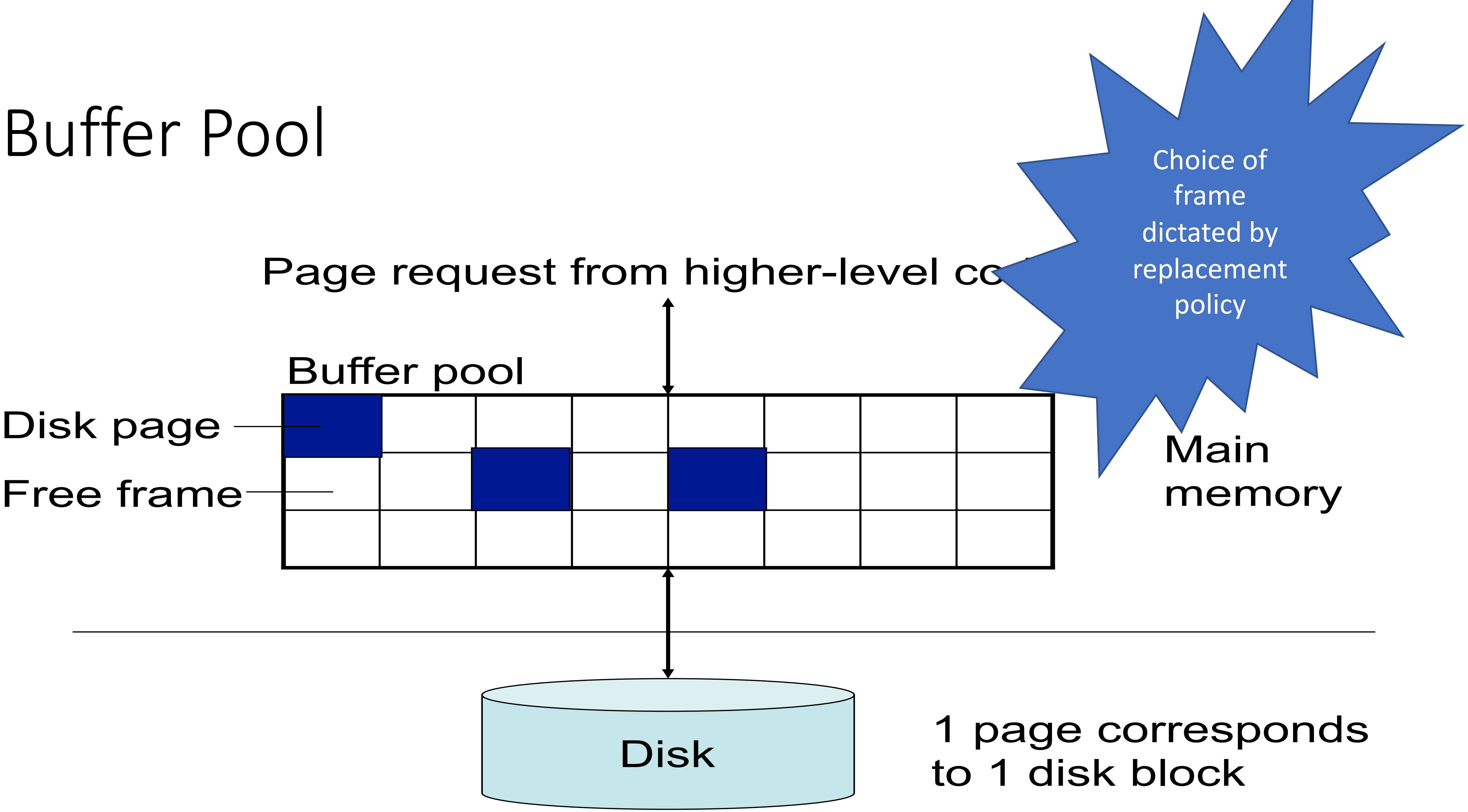
Types of Failures

- Type of Crash Prevention
- Wrong data entry
  - Constraints and Data cleaning
- Disk crashes Redundancy:
  - e.g. RAID, archive
- Data center failures
  - Remote backups or replicas
- System failures:
  - e.g. power DATABASE RECOVERY

System Failures

- Each transaction has internal state
  - When system crashes, internal state is lost
- Don’t know which parts executed and which didn’t
  - Need ability to undo and redo

Buffer Pool


Page request from higher-level co
Buffer pool
Disk page
Free frame
Main memory
Disk
1 page corresponds to 1 disk block
Choice of frame dictated by replacement policy

- Enables higher layers of the DBMS to assume that needed data is in main memory
  - Caches data in memory.
- Problems when crash occurs:
  - If committed data was not yet written to disk
  - If uncommitted data was flushed to disk

A model for transactions

- Database state: The space of disk blocks holding the database elements
- Buffer manager: The virtual or main memory address space
- Transaction state: The local address space of the transaction

Primitive Operations

- READ(X,t)
  - copy value of data item X to transaction local variable t
- WRITE(X,t)
  - copy transaction local variable t to data item X
- INPUT(X)
  - read page containing data item X to memory buffer
- OUTPUT(X)
  - write page containing data item X to disk

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

Example

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)					
t = t* 2					
WRITE(A,t)					
INPUT(B)					
READ(B,t)					
t = t* 2					
WRITE(B,t)					
OUTPUT(A)					
OUTPUT(B)					
COMMIT					

Example

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2					
WRITE(A,t)					
INPUT(B)					
READ(B,t)					
t = t* 2					
WRITE(B,t)					
OUTPUT(A)					
OUTPUT(B)					
COMMIT					

Example

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)					
INPUT(B)					
READ(B,t)					
t = t* 2					
WRITE(B,t)					
OUTPUT(A)					
OUTPUT(B)					
COMMIT					

Example

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)					
READ(B,t)					
t = t* 2					
WRITE(B,t)					
OUTPUT(A)					
OUTPUT(B)					
COMMIT					

Example

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)					
t = t* 2					
WRITE(B,t)					
OUTPUT(A)					
OUTPUT(B)					
COMMIT					

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

Example

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t = t* 2					
WRITE(B,t)					
OUTPUT(A)					
OUTPUT(B)					
COMMIT					

Example

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t = t* 2	16	16	8	8	8
WRITE(B,t)					
OUTPUT(A)					
OUTPUT(B)					
COMMIT					

Example

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t = t* 2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)					
OUTPUT(B)					
COMMIT					

Example

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t = t* 2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)					
COMMIT					

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

Example

	Txn State	Buffer Pool		Disk State	
Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t = t* 2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16
COMMIT					

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

Example: is this bad?

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t = t* 2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16
COMMIT					

Yes A = 16, B = 8

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

Example: is this bad?

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t = t* 2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16
COMMIT					

Yes A = 16, B = 16
But not committed

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

Example: is this bad?

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t = t* 2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16
COMMIT					

No; DB is consistent

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

Example (Output after commit)

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t = t* 2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Example:

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

	Txn State	Buffer Pool		Disk State	
Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t = t* 2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Atomic Transactions

- FORCE or NO-FORCE
  - Should all updates of a transaction be forced to disk before the transaction commits?
- STEAL or NO-STEAL
  - Can an update made by an uncommitted transaction overwrite the most recent committed value of a data item on disk?

Force/No-Steal
(Most strict)

- FORCE: Pages of committed transactions must be forced to disk before commit
- NO-STEAL: Pages of uncommitted transactions cannot be written to disk

No-force/Steal (least strict)

- NO-FORCE: Pages of committed transactions need not be written to disk
- STEAL: Pages of uncommitted transactions may be written to disk

- In both cases, need a Write Ahead Log (WAL) to provide atomicity in face of failures

Write-ahead Log (WAL)

- The Log: append-only file containing log records
  - Records every single action of every TXN
  - Forces log entries to disk as needed
  - After a system crash, use log to recover
- Three types: UNDO, REDO, UNDO-REDO

Policies and Logs

Most strict

	STEAL	NO-STEAL
FORCE	Undo Log	Lab4
NO-FORCE	Undo-redo Log	Redo Log

Least strict

“Undo” Log

- FORCE and STEAL

Undo Logging

- Log records
  - <START, T>
    - transaction T has begun
  - <COMMIT, T>
    - T has committed
  - <ABORT, T>
    - T has aborted
  - <T,X,v>
    - T has updated element X, and its old value was v
    - Idempotent, physical log records

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

Example:

	Txn State	Buffer Pool		Disk State		
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**	**UNDO Log**
						<START, T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t = t* 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t = t* 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT,T>

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

Example:

	Txn State	Buffer Pool		Disk State		
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**	**UNDO Log**
						<START, T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t = t* 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t = t* 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT,T>

We UNDO by setting A= 8, B= 8

Example:

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

	Txn State	Buffer Pool		Disk State		
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**	**UNDO Log**
						<START, T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t = t* 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t = t* 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT,T>

Nothing to be done COMMIT!

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

Example:

	Txn State	Buffer Pool		Disk State		
Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START, T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t = t* 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t = t* 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT,T>

After Crash

Disk A	Disk B
8	16

<START, T>
<T,A,8>
<T,B,8>

After Crash

Disk A	Disk B
8	16

<START, T>
<T,A,8>
<T,B,8>

Q: Which direction to undo the actions?

After Crash

Disk A	Disk B
8	16

<START, T>
<T,A,8>
<T,B,8>

Q: Which direction to undo the actions?

A: In UNDO log, we start at the most recent and go backwards in time

After Crash

Disk A	Disk B
8	8

<START, T>
<T,A,8>
<T,B,8>

Q: Which direction to undo the actions?

A: In UNDO log, we start at the most recent and go backwards in time

After Crash

Disk A	Disk B
8	8

<START, T>
<T,A,8>
<T,B,8>

Q: Which direction to undo the actions?

A: In UNDO log, we start at the most recent and go backwards in time

- If we see NO Commit statement:
  - We UNDO both changes: A=8, B=8 •
  - The transaction is atomic, since none of its actions have been executed
- If we see that T has a Commit statement
  - We don’t undo anything •
  - The transaction is atomic, since both it’s actions have been executed

Recovery Manager

- After system’s crash, run recovery manager
  - Decide for each transaction T whether it is completed or not
  - <START>…..<COMMIT> …..…. = yes
  - <START>…..<ABORT> …..…. = yes
  - <START>………..…. = no

- Undo all modifications by *incomplete* transactions

Read log <u>from the end;</u>
cases: :
<COMMIT,T> mark T as completed
<ABORT, T> : mark T as completed
<T, X, v>: if T is not completed
then write X=v to disk
else ignore
<START, T>: ignore

Recovery with Undo Log

- Which updates are undone ?
- How far back do we need to read in the log?
- What happens if second crash during recovery?

…
…
<T6, X6, V6>
…
…
<START, T5>
<START, T4>
<T1, X1, v1>
<T4,X4, v3>
<T5,X5, v1>
<COMMIT, T5>
<T3,X1, v1>
<T2,X1, v1>

Recovery with Undo Log

- Which updates are undone ?
  - All excep t T5
- How far back do we need to read in the log?
  - To the beginning
- What happens if second crash during recovery?
  - Idempotent.

…
…
<T6, X6, V6>
…
…
<START, T5>
<START, T4>
<T1, X1, v1>
<T4,X4, v3>
<T5,X5, v1>
<COMMIT, T5>
<T3,X1, v1>
<T2,X1, v1>

Policies and Logs

Most strict

	STEAL	NO-STEAL
FORCE	Undo Log	Lab4
NO-FORCE	Undo-redo Log	Redo Log

Least strict

Recovery with Undo Log

- When must we force pages to disk ?

- RULES: log entry before OUTPUT before COMMIT

Recovery with Undo Log: FORCE Rules

- U1: If T modifies X, then <T, X, v> must be written to disk before OUTPUT(X)
- U2: If T commits, then OUTPUT(X) must be written to disk before <COMMIT, T>
- Hence: OUTPUTs are done early, before the transaction commits

REDO

- NO-FORCE and NO-STEAL

- One minor change to the undo log:
- <T,X,v> = T has updated element X, and its new value is v

Example:

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t = t* 2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

Is this bad?

	Txn State	Buffer Pool		Disk State	
Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t = t* 2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t = t* 2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

No-Steal Redo Logging Rules

- R1: If T modifies X, then both <T,X,v> and <COMMIT, T> must be written to disk before OUTPUT(X)
- Hence: OUTPUTs are done late

Undo/Redo Logging

- Undo logging:
  - OUTPUT must be done early
  - If <COMMIT, T> is seen, T definitely has written all its data to disk (hence, don’t need to redo) – inefficient
- Redo logging
  - OUTPUT must be done late
  - If <COMMIT, T> is not seen, T definitely has not written any of its data to disk (hence there is not dirty data on disk, no need to undo) – inflexible
- Would like more flexibility on when to OUTPUT: undo/redo logging