Fundamentals of Database Systems

Transaction Management

Jef Wijsen

University of Mons-Hainaut (UMH)

Reference

Chapters 8, 9, and 10 of:

H. Garcia-Molina, J.D. Ullman, J. Widom: *Database System Implementation.* Prentice Hall, 2000.

Transaction Management

PART I:

Recovery From System Failures

TOC
Recovery From System Failures

- Transactions
  - General properties
  - Database buffering
  - Effect of system failure
- Logging to recover from system failure
- Undo/Redo logging
- Undo logging
- Redo logging
- Finale

Transactions

General properties

Transaction (Trx)

- A transaction is an execution of a program, and corresponds to a logical unit of work.
- We focus on short-running trx, for example:

  banking: **transfer-money(from,to,amount)**

  airline reservations: **reserve-seat(passenger,flight)**
- Running example:

  BEGIN; A := A+B; B := 0; END;
- A and B are database elements.
  Think of A and B as the number of Euros owned by An and Bob resp. The trx gives Bob's money to An.
- DB elements are persistent (no initialization needed).

SQL

```
BEGIN
UPDATE WEALTH
SET SUM=SUM+(SELECT SUM
                    FROM WEALTH
                    WHERE NAME='Bob')
WHERE NAME='An';

UPDATE WEALTH
SET SUM=0
WHERE NAME='Bob';
END
```

WEALTH

NAME	SUM
An	2
Bob	1

A

B

ACID Properties

- Atomicity: Transactions are all-or-nothing.
- Consistency: We will assume that transactions preserve database consistency.
- Isolation: Inconsistent intermediate data must be concealed from all the rest.
- Durability: Once a trx has completed, its updates survive, even if there is a subsequent system crash.

ACID By Example (1)

	A	B	
BEGIN	2	1	←Initial state
A:=A+B	3	1	← Intermediate
B:=0	3	0	← Final state
END			

ACID By Example (2)

- Atomicity: The transaction should not prematurely end, leaving the inconsistent state A=3, B=1.
- Consistency: The final state A=3, B=0 is consistent with the initial state A=2, B=1.
- Isolation: The intermediate state A=3, B=1 is inconsistent and should not be visible to other transactions.
- Durability: The final state A=3, B=0 must not be lost in a subsequent system crash.

Transaction Management

- Ensuring *Isolation:*

  Conflicts arise because multiple concurrent transactions ‘compete’ for accessing the same data.

- Ensuring *Atomicity and Durability:*

  Failures (program error, electricity failure, disk crash, explosion,…) may erase data or cause a trx to end prematurely.

  Imagine your bank telling you that your account was definitely lost in a disk crash…

- Transaction management deals with concurrency control and recovery from failure.

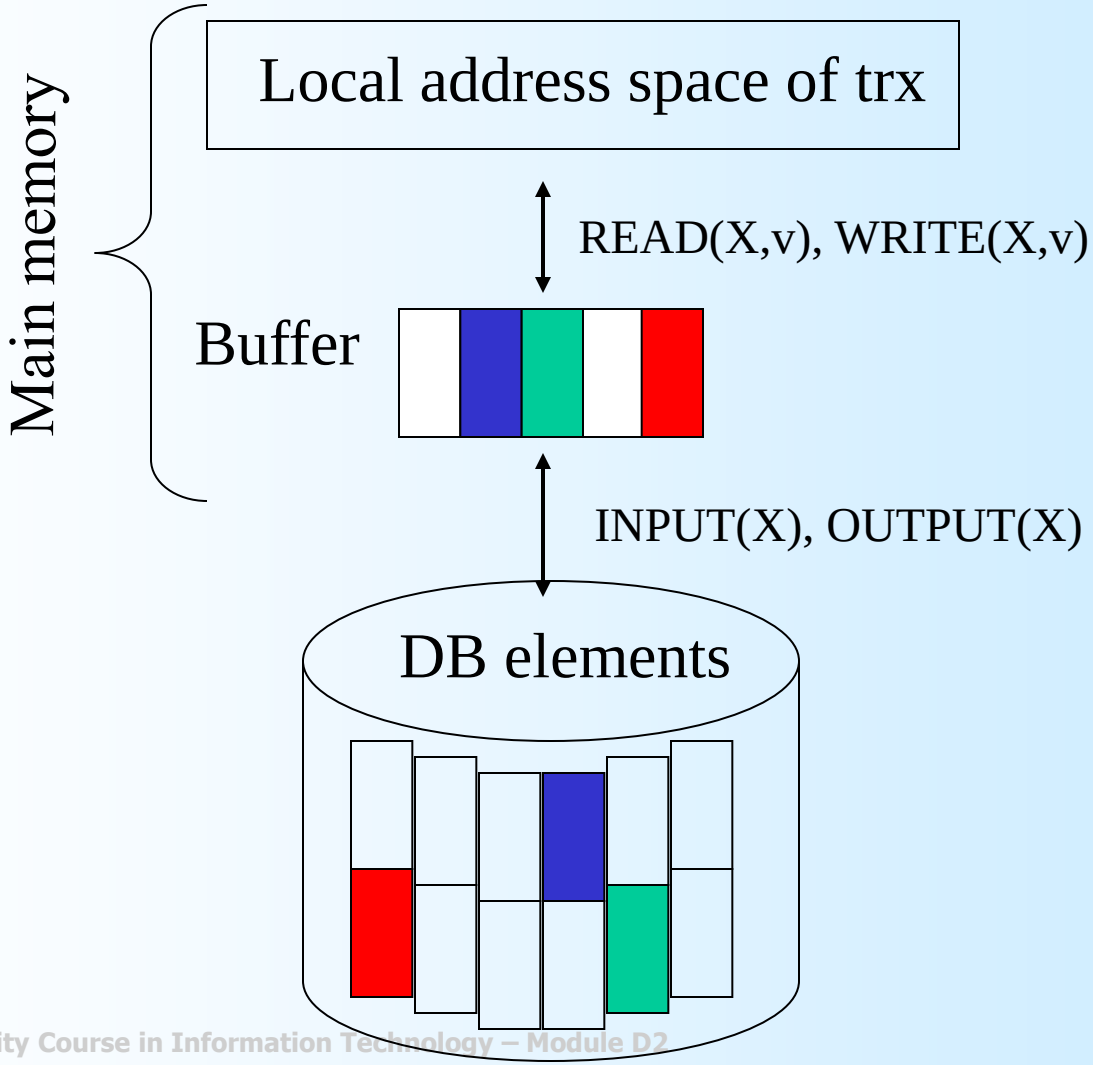
Failures

- System failures (e.g., power outage), which affect all trx currently in progress but do not physically damage the database. **The contents of main memory are lost.**
- Media failures (e.g., head crash on the disk), which do cause damage to the database.
- Our focus will be on recovery from system failures.

Transactions

Database buffering

Address Spaces



Primitive Operations

- INPUT(X): Copy the disk block containing database element X to a buffer block.
- READ(X,v): Copy X to the trx's local variable v (entails INPUT(X) if X is not in buffer).
- WRITE(X,v): Copy the value of v to X in buffer (entails INPUT(X) if X is not in buffer).
- OUTPUT(X): Copy the buffer block containing X to disk.
- READ and WRITE are issued by trx.

  Importantly, INPUT and OUTPUT are not trx commands, but are issued by DBMS modules (trx manager, buffer manager, log manager).

Our Trx Revisited

- So

BEGIN
A:=A+B
B:=0
END

could be expressed as:

BEGIN
READ(A,v)
READ(B,w)
v:=v+w
WRITE(A,v)
WRITE(B,0)
END

- In addition, we could show OUTPUT steps, even though these are not the responsibility of the trx *per se*.

What is a Database Element?

- Database elements can be tuples (e.g., tuple about An).
- The primitive operations assume that database elements reside within a single disk block…
- We will also assume that no block contains more than one database element (cf. discussion later).
- That would be true for database elements that *are* blocks.

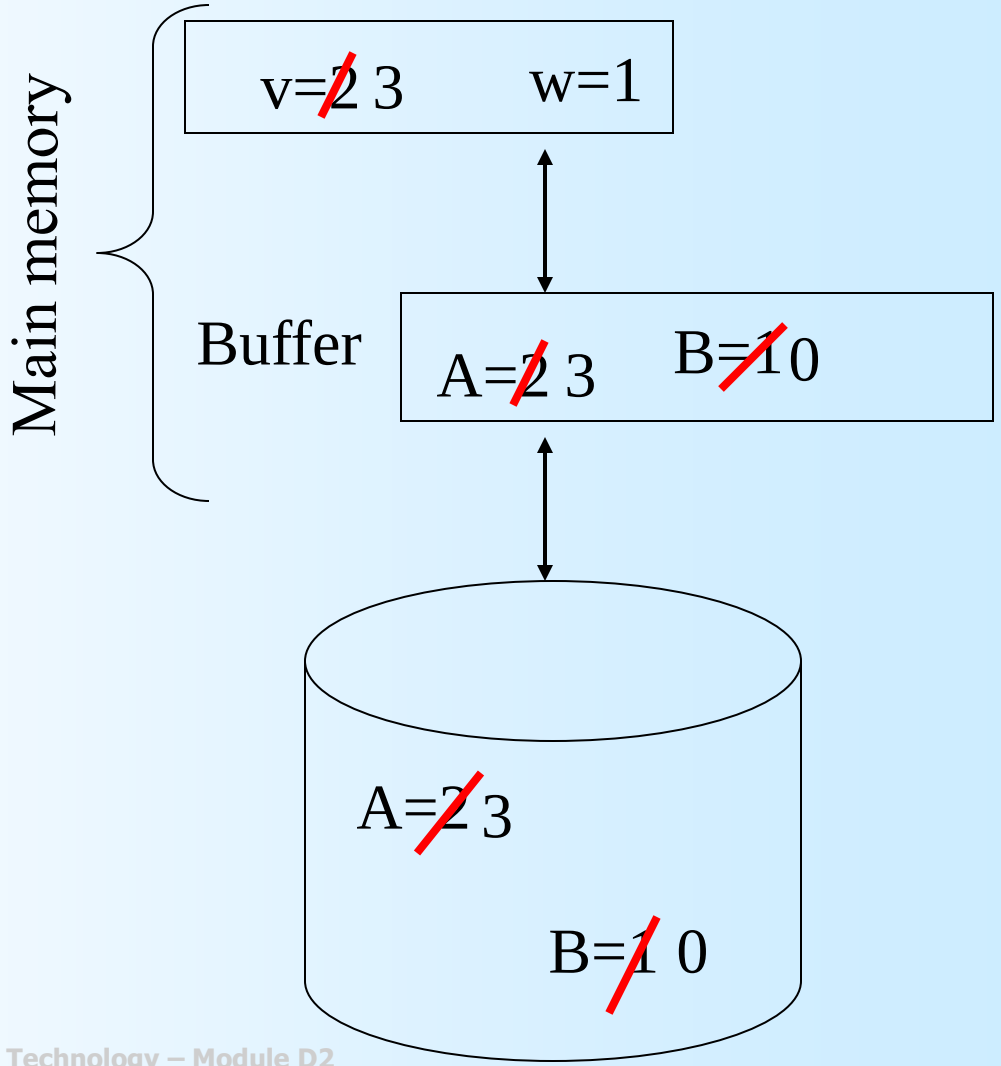
Effect of Trx (1)

Disk Block, or DataBase

Action	v	w	Buf A	Buf B	DB A	DB B
BEGIN					2	1
READ(A,v)	2		2		2	1
READ(B,w)	2	1	2	1	2	1
v:=v+w	3	1	2	1	2	1
WRITE(A,v)	3	1	3	1	2	1
WRITE(B,0)	3	1	3	0	2	1
OUTPUT(A)	3	1	3	0	3	1
END			3	0	3	1
OUTPUT(B)	3	1	3	0	3	0

Effect of Trx (2)

READ(A,v)
READ(B,w)
v:=v+w
WRITE(A,v)
WRITE(B,0)
OUTPUT(A)
OUTPUT(B)

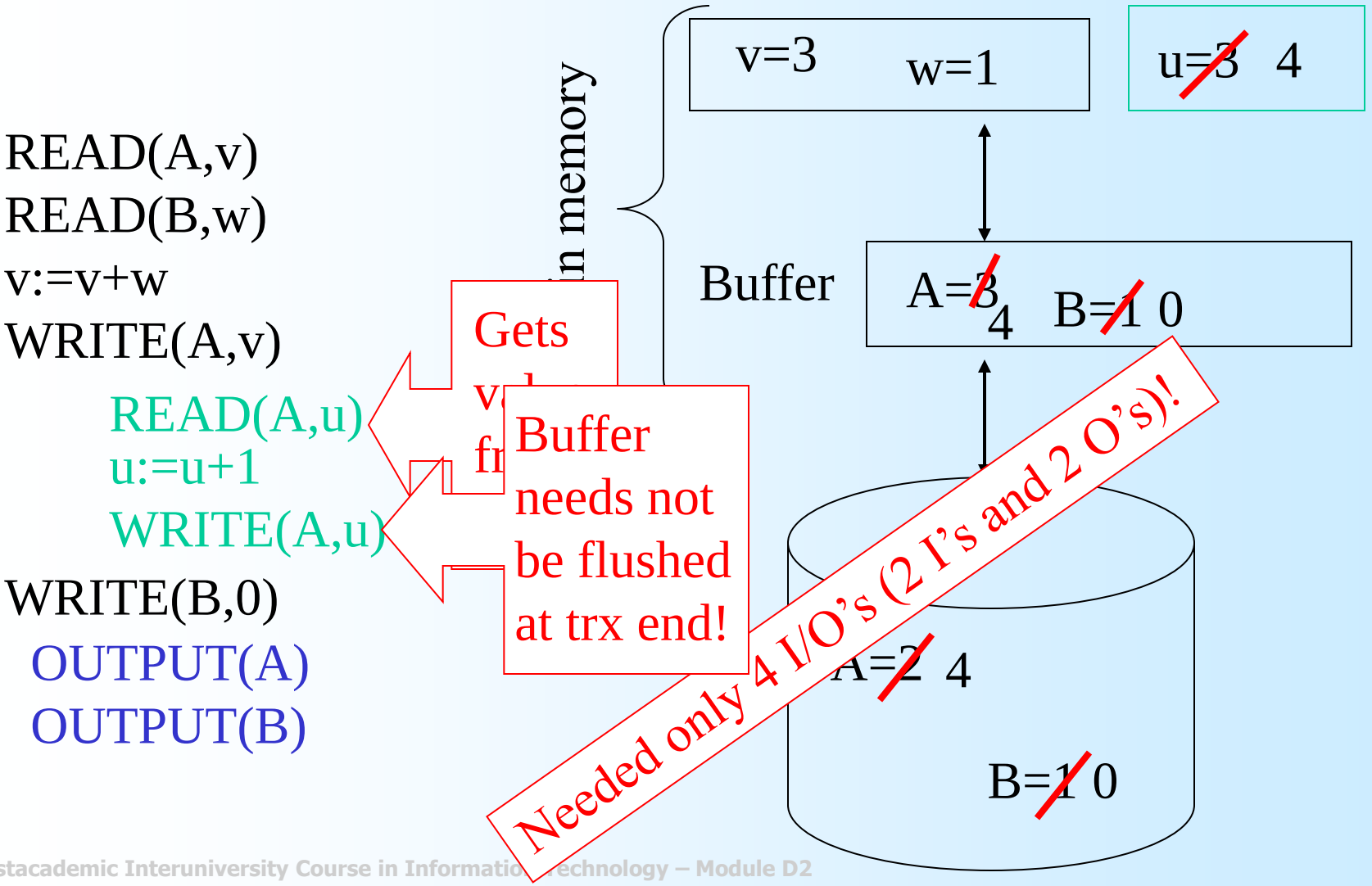


Concurrent Transactions (1)

BEGIN	Second trx adds 1 to Ann's account.
READ(A,v)	
READ(B,w)	
v:=v+w	
WRITE(A,v)	
	BEGIN
	READ(A,u)
	u:=u+1
	WRITE(A,u)
	END
WRITE(B,0)	
OUTPUT(A)	
END	
OUTPUT(B)	

- OUTPUT is initiated by buffer manager on behalf of both trx.
- Both trx issue a WRITE of A, but there is only a single OUTPUT of A.

Concurrent Transactions (2)



A Note on Interleaving and Isolation

- You may think that interleaving conflicts with isolation…

T	U
READ(A,v)	
READ(B,w)	
v:=v+w	
WRITE(A,v)	
	READ(A,u)
	READ(B,v)
WRITE(B,0)	

U reads new A and old B, violating isolation…

T	U
READ(A,v)	
READ(B,w)	
v:=v+w	
WRITE(A,v)	
	READ(A,u)
WRITE(B,0)	
	READ(B,v)

U reads new A and new B, satisfying isolation…

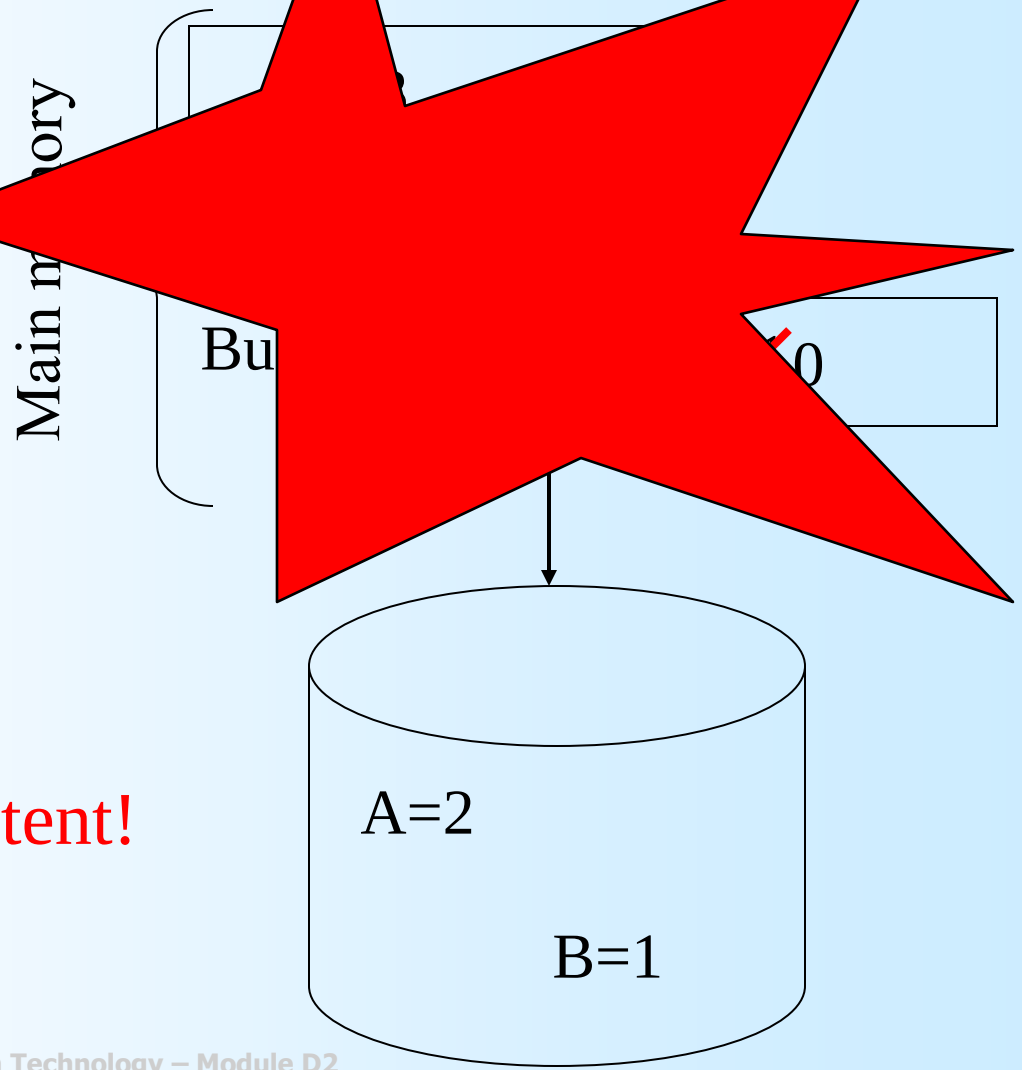
- Characterizing admissible interleavings is at the center of concurrency control.

Transactions

Effect of system failure

Effect of System Failure (1)

READ(A,v)
READ(B,w)
v:=v+w
WRITE(A,v)
WRITE(B,0)



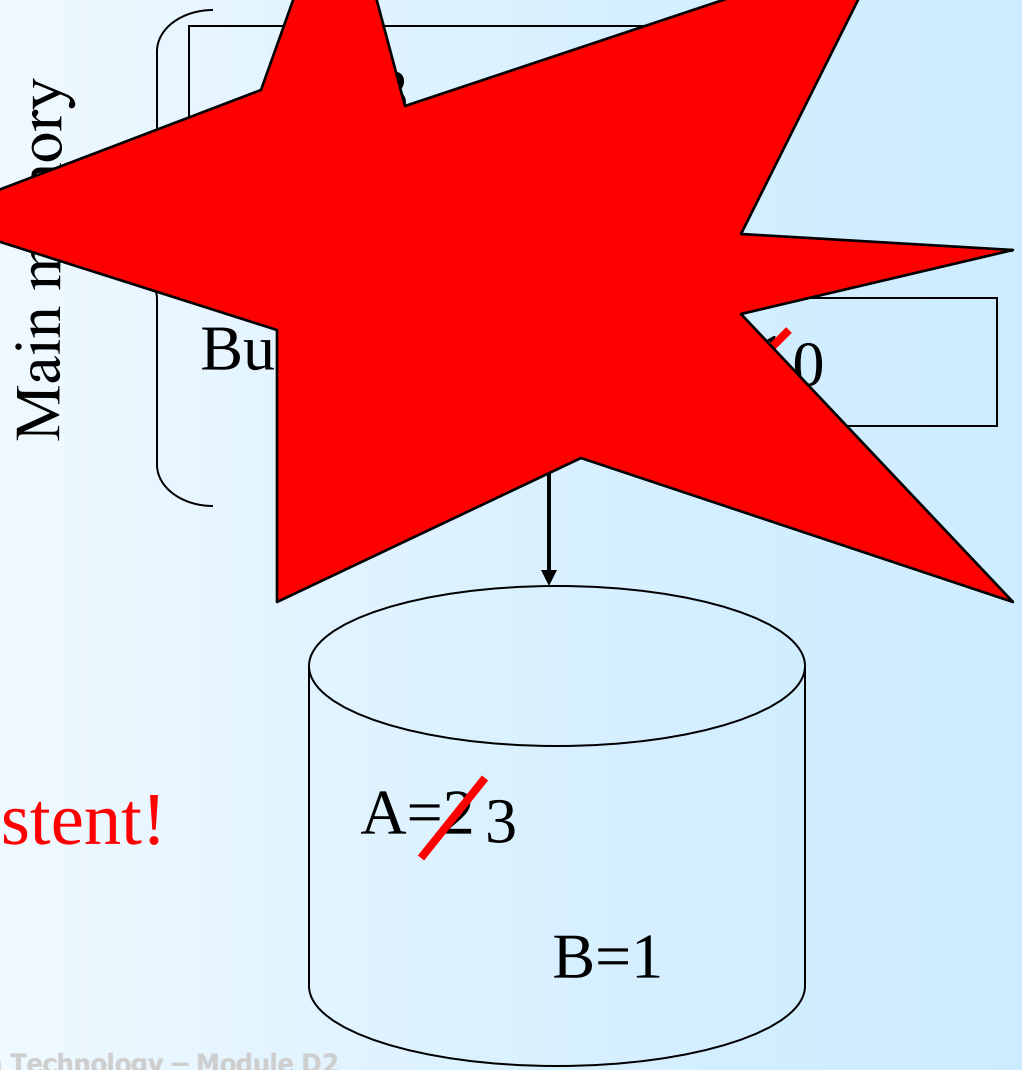
The database is consistent!

Effect of System Failure (2)

- Recall that the buffer manager issues OUTPUT operations, which need not be synchronized with WRITE or END operations.
- Recall also that main-memory does not survive a system failure.
- If a system error occurs before any OUTPUT operation is executed, then there is no effect to the database stored on disk. The transaction would be all-or-nothing (actually nothing).

Effect of System Failure (3)

READ(A,v)
READ(B,w)
v:=v+w
WRITE(A,v)
WRITE(B,0)
OUTPUT(A)



The database is inconsistent!

Effect of System Failure (4)

- If there is a system error after OUTPUT(A) but before OUTPUT(B), then the database is left in an inconsistent state. At recovery time, we should either ‘roll back’ to A=2, B=1, or ‘roll forward’ to A=3, B=0.
- However, from the database we cannot tell:
  - which trx were active at the time the failure occurred;
  - what values need to be restored to make the database consistent.
- Solution: logging database changes.

TOC
Recovery From System Failures

- Transactions
- Logging to recover from system failure
- Undo/Redo logging
- Undo logging
- Redo logging
- Finale

Logging

- A log is a sequence of log records, each telling something about what some trx has done.

Action by trx T	Log record
BEGIN	[START T]
WRITE(X,v)	[T, X, ⟨before_image⟩, ⟨after_image⟩]
END	[COMMIT, T]

- The log is a file opened for appending only.
- The log manager can issue a FLUSH LOG command to tell the buffer manager to copy log blocks to disk.
- Log records are flushed in the order written.

Log Entries

Action	v	w	Buf A	Buf B	DB A	DB B	Log
BEGIN					2	1	[START T]
READ(A,v)	2		2		2	1	
READ(B,w)	2	1	2	1	2	1	
v:=v+w	3	1	**2**	1	2	1	
WRITE(A,v)	3	1	**3**	**1**	2	1	[T,A,**2,3**]
WRITE(B,0)	3	1	3	**0**	2	1	[T,B,**1,0**]
OUTPUT(A)	3	1	3	0	3	1	
END			3	0	3	1	[COMMIT T]
OUTPUT(B)	3	1	3	0	3	0	

Transaction Commit

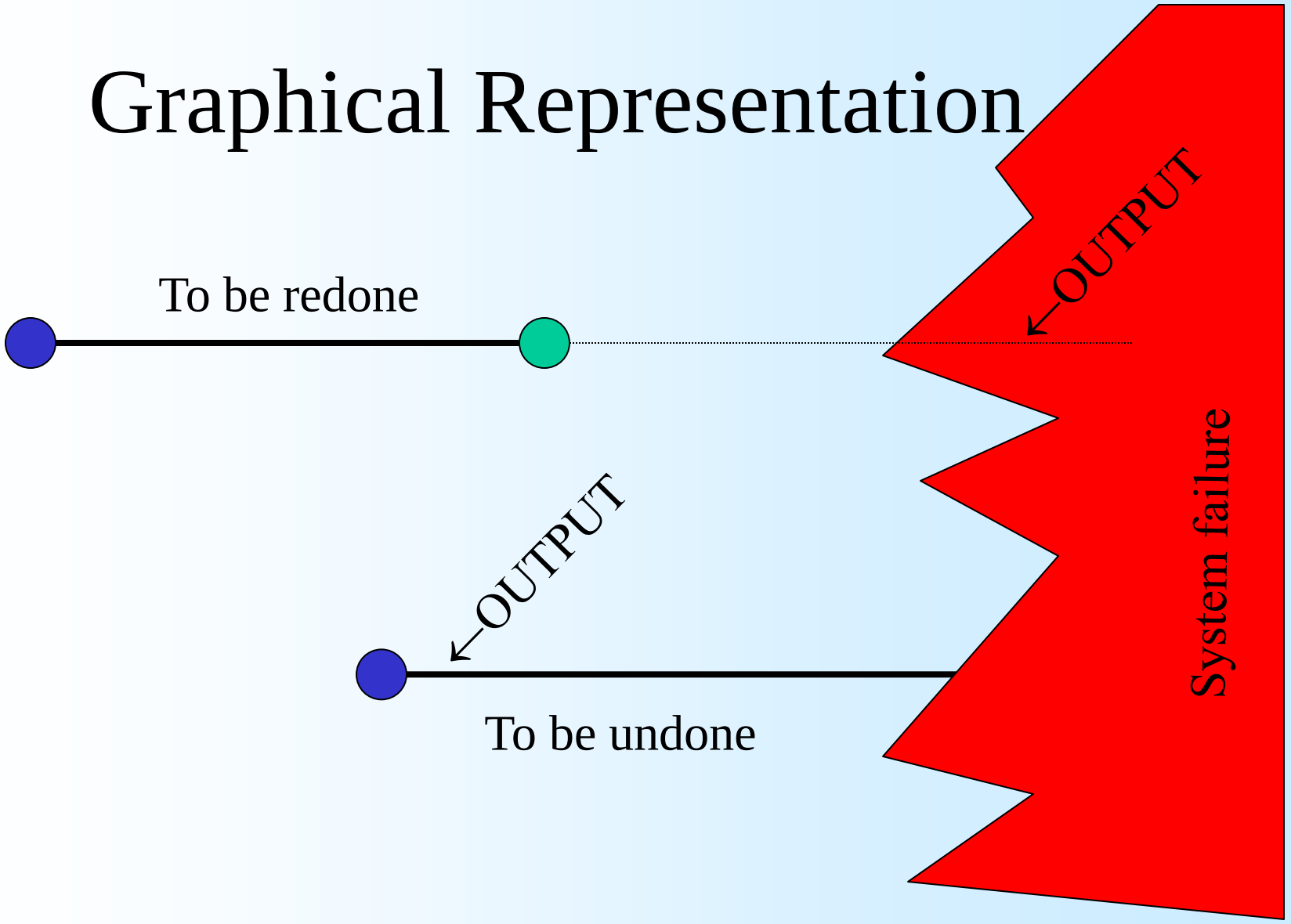
- We say that a transaction T is committed if a [COMMIT T] record appears in the log **on disk**.
- Importantly, database buffers may or may not have been copied to disk by OUTPUT actions at commit time; that decision is the responsibility of the buffer manager in general.
- As a matter of fact, in order to reduce the number of disk I/O's, database systems can and will allow a change to exist only in volatile main-memory storage.

Roll Back or Roll Forward?

Two categories of trx need to be distinguished during recovery from system failure:

- Committed transaction with some (or all) of its changes not on disk. Durability implies that such transaction has to be redone (‘rolled forward’).
- Uncommitted transaction with some (or all) of its changes on disk. Atomicity implies that such transaction has to be undone (‘rolled back’).

Graphical Representation



TOC
Recovery From System Failures

- Transactions
- Logging to recover from system failure
- Undo/Redo logging
  - The rules
  - The recovery procedure
  - Checkpointing
  - Exercise
- Undo logging
- Redo logging
- Finale

Undo/Redo Logging

The rules

Rule for Undo/Redo Logging

- *Rule UR1:* Before modifying any database element X on disk because of changes made by some transaction T, it is necessary that the update record

  [T, X, ⟨before_image⟩, ⟨after_image⟩]

  appear on disk.
- That is, between WRITE(X,v) and a following OUTPUT(X), there must be a FLUSH LOG.

Need For ⟨before_image⟩

- The flushed ⟨before_image⟩ will be needed if T has to be undone later on:
  1. T issues WRITE(X,v)
  2. the log is flushed
  3. the buffer manager issues OUTPUT(X)
  4. a system failure occurs before T can commit
- T has to be undone. The ⟨before_image⟩ of X is available in the log (not in the database!)

Need For ⟨after_image⟩

- The flushed ⟨after_image⟩ will be needed if T has to be redone later on:
  1. T issues WRITE(X,v)
  2. T executes END
  3. The log is flushed, hence T is now committed
  4. a system failure occurs before the buffer manager has issued OUTPUT(X)
- T has to be redone. The ⟨after_image⟩ of X is available in the log (not in the database!)

Rule *UR1* Example

If a system failure occurs after OUTPUT(A) but before commit, A has to be reset to its before-image 2.

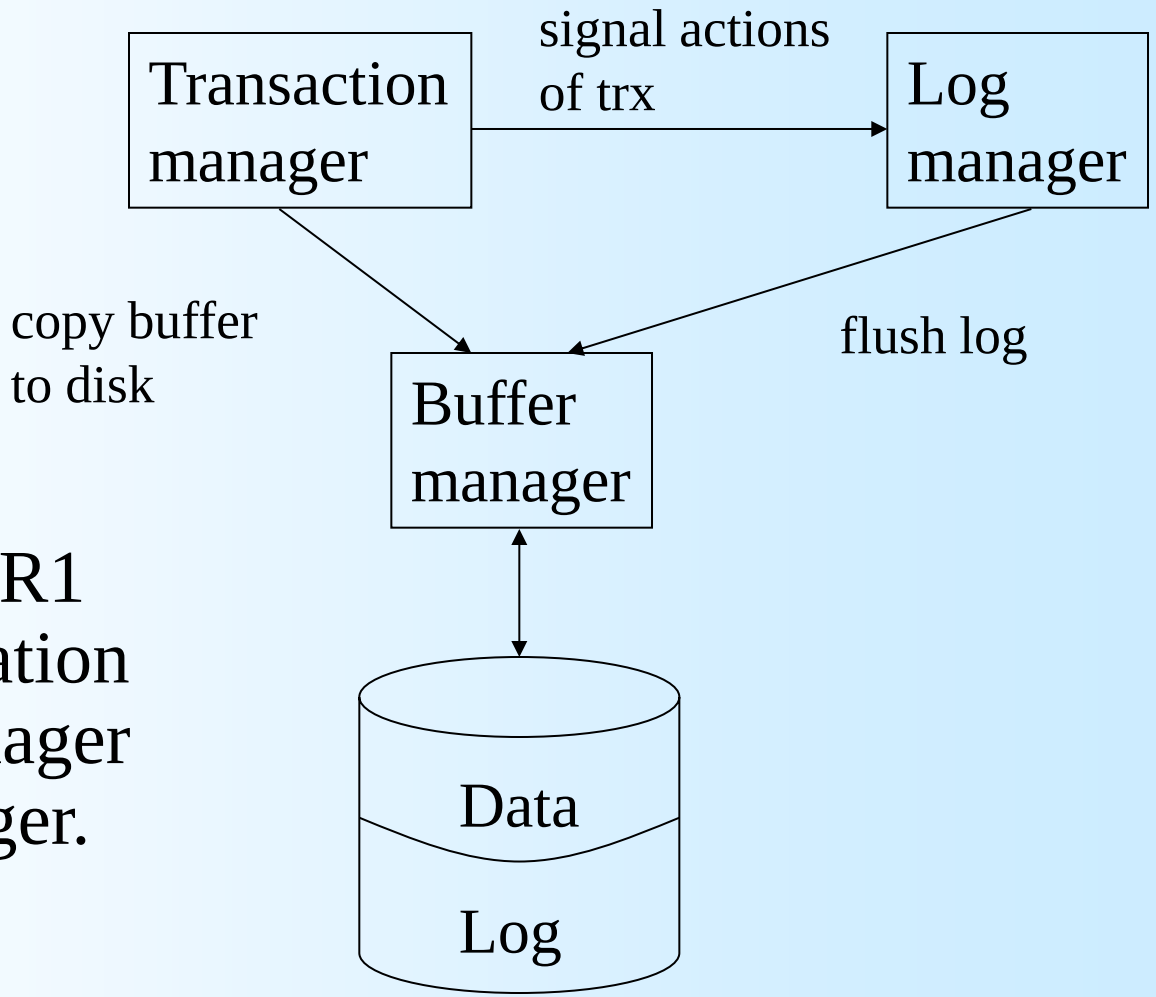
If a system failure occurs after commit but before OUTPUT(B), B has to be set to its after-image 0.

The log must be flushed in between.

The log must be flushed in between.

Action	Log
BEGIN	[START T]
READ(A,v)	
READ(B,w)	
v:=v+w	
WRITE(A,v)	[T,A,2,3]
WRITE(B,0)	[T,B,1,0]
OUTPUT(A)	
END	[COMMIT T]
OUTPUT(B)	

Transaction Manager



Enforcing rule UR1 requires coordination between log manager and buffer manager.

Problem With Delayed Commit

- Assume a system crash after END of trx T but before [COMMIT T] is flushed to disk.
- Trx T appears to the user to have been completed, but is to be undone during system recovery.
- It is therefore advisable to flush a [COMMIT T] record to disk as soon as it appears in the log.

Undo/Redo Logging

The recovery procedure

How to recover from

[START T][T,A,2,3]?

- All we know is that the system failure occurred after WRITE(A,v) but before OUTPUT(B).
- Indeed, since [T,B,1,0] has not reached the log on disk, OUTPUT(B) has not been executed (Rule *UR1*).
- OUTPUT(A) may or may not have been executed.
- Anyway, the recovery procedure undoes T by writing 2 for the database element A.

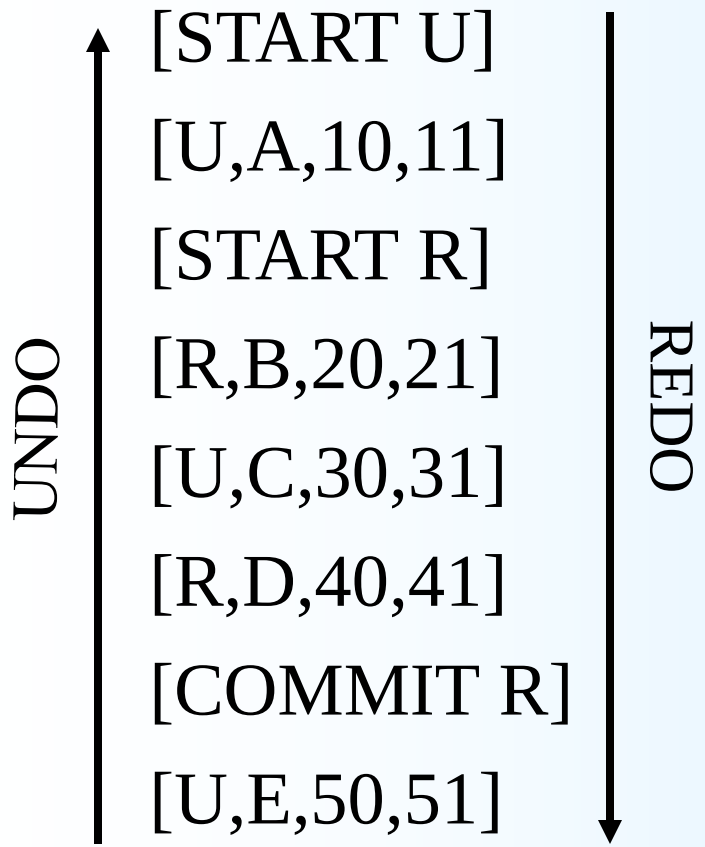
How to recover from [START T][T,A,2,3][T,B,1,0][COMMIT T]?

- We know that the system failure occurred after the action END.
- OUTPUT(B) may or may not have been executed.
- Anyway, the recovery procedure redoes T by writing 3 for the database element A, and 0 for B.

Recovery With Undo/Redo Logging (in general)

- Identify the committed and uncommitted trx.
- Undo all the uncommitted trx in the order latest-first (i.e., backward).
- Redo all the committed trx in the order earliest-first (i.e., forward).

Example



- U is to be undone,
  R is to be redone.
- Undo U.
  E := 50
  C:= 30
  A:=10
- Redo R.
  B:=21
  D:=41

Notes

- You will realize that redoing T does not mean that the program of T is re-executed.
- Trx that are undone may have to be restarted later on, but that decision is outside the scope of trx management *per se*.

Undo/Redo Logging

Checkpointing

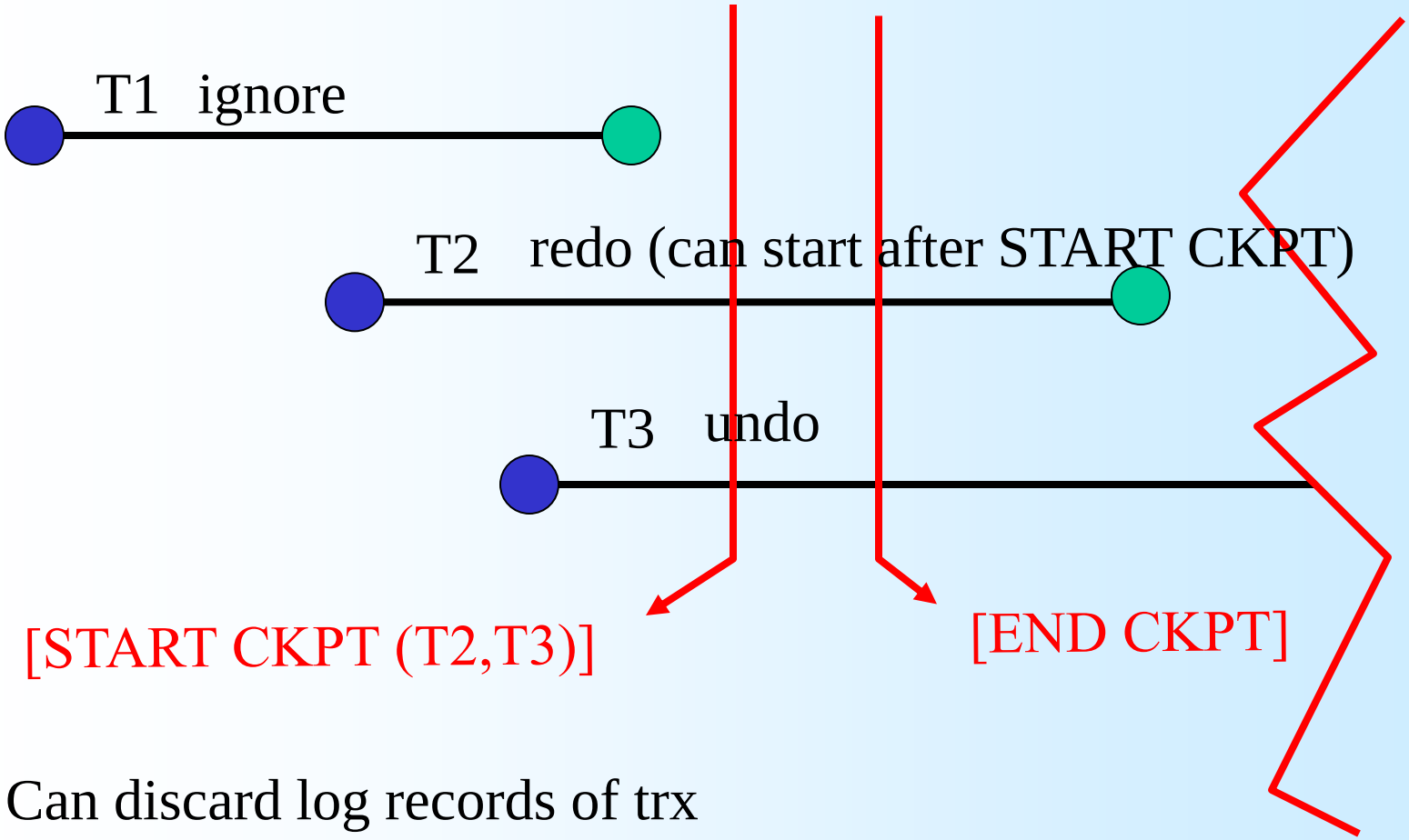
Checkpointing

- After a system failure, we have to redo all trx that ever committed, because their updates may not have reached the database on disk.
- Hence, log parts cannot be discarded, and the log keeps growing.
- To alleviate this problem, we periodically take a checkpoint: write to disk all modified buffers and record this in the log.

Checkpointing an Undo/Redo Log

- Write a [START CKPT (T_1,…,T_k)] record to the log, where T_1,…,T_k are the active trx, and flush the log.
- Write to disk all buffers that are dirty; i.e., they contain one or more changed data elements.
- Write an [END CKPT] record to the log, and flush the log.

Effect of Checkpoint



Can discard log records of trx
that committed before START CKPT.

Undo/Redo Logging

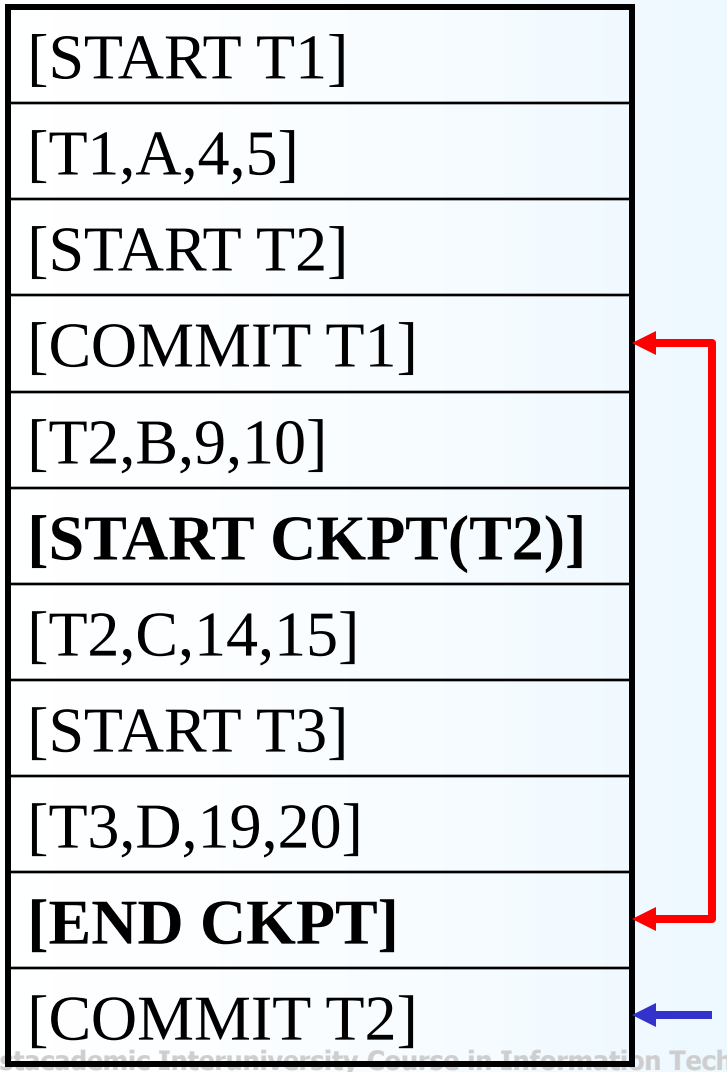
Exercise

Exercise 1

- The following is the content of an **undo/redo** log after a crash. Describe the changes to disk that have to be made by the recovery manager.

[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]

Exercise 1 (Cntd.)



The uncommitted trx T3 has to be undone.

All updates of T1 have reached the database.

T2 has to be redone.

- The recovery manager will set D to 19 and C to 15.
- It is not necessary to set B to 10, as we flush B to disk during the checkpoint.

TOC
Recovery From System Failures

- Transactions
- Logging to recover from system failure
- Undo/Redo logging
- Undo logging
  - The rules
  - The recovery procedure
  - Checkpointing
  - Exercise
- Redo logging
- Finale

Undo Logging

The rules

Undo/No-Redo Logging

- Redo-work is needed because updates of committed trx may not have reached disk.
- Undo/No-Redo logging requires that no trx commit before its updates have reached disk. (You may find this quite natural!)
- Advantage: avoids redo-work, and hence the need for after-images.
- WRITE(X,v) is logged as [T,X,⟨before_image⟩].
- Undo logging = Undo/No-Redo logging.

Rules for Undo Logging (1)

- *Rule U1:* If transaction T modifies database element X, then the log record of the form [T,X,⟨before_image⟩] must be written to disk before the new value of X is written to disk (cf. rule *UR1*).
- *Rule U2:* If a transaction commits, then its COMMIT log record must be written to disk only after all database elements changed by the transaction have been written to disk, but as soon thereafter as possible.

Rules for Undo Logging (2)

- Hence, material associated with one trx must be written to disk in the following order:
    1. The log records indicating changed database elements.
    2. The changed database elements themselves.
    3. The COMMIT log record.
- The order of (1) and (2) applies to each database element individually, not to the group of update records for a trx as a whole.

Disadvantage of Undo Logging

- Undo logging requires that data be written to disk before the trx commits, perhaps increasing the number of disk I/O's that need to be performed.

Undo Logging

The recovery procedure

Recovery Using Undo Logging

- Basically like in Undo/Redo logging with Redo removed.
- The recovery manager scans the log backward from the end, remembering [COMMIT T] records.

  As it sees a record [T,X,⟨before_image⟩], then:

  - If T is a trx whose COMMIT record has been seen, then do nothing.
  - Otherwise, T is an uncommitted trx. The recovery manager must change the value of X in the database to ⟨before_image⟩.

Undo Logging

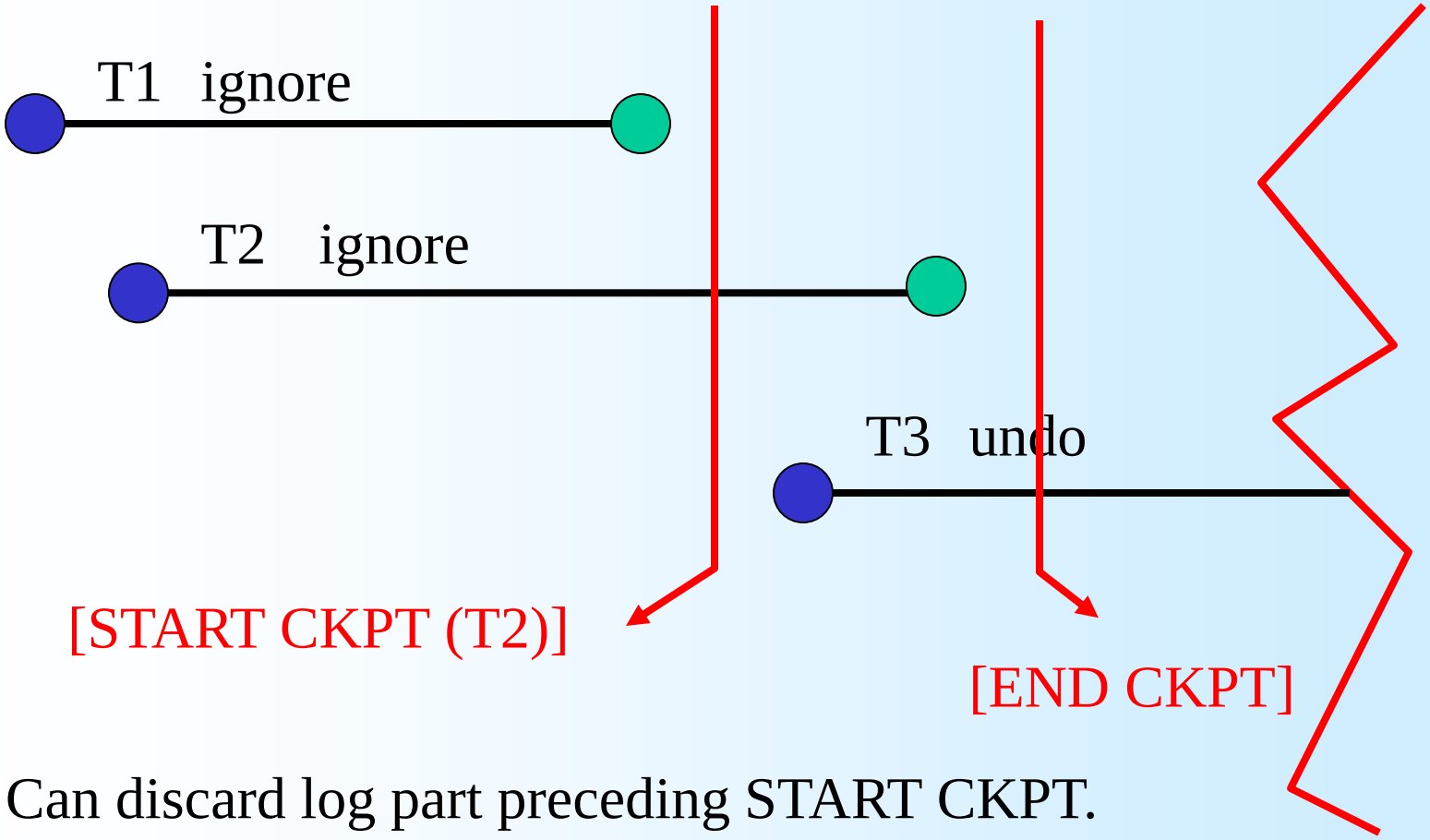
Checkpointing

Checkpointing an Undo Log (1)

- Log records of T can be discarded once T has committed!
- Checkpointing here means waiting until currently active trx have completed:

1. Write a [START CKPT (T_1,…,T_k)] record to the log, where T_1,…,T_k are the active trx, and flush the log.
2. Wait until T_1,…,T_k commit.
3. When all T_1,…,T_k have completed, write a log record [END CKPT] and flush the log.

Checkpointing an Undo Log (2)



Can discard log part preceding START CKPT.

Undo Logging

Exercise

Exercise 2

- The following is the content of an **undo** log after a crash. Describe the changes to disk that have to be made by the recovery manager.

[START T1]
[T1,A,5]
[START T2]
[T2,B,10]
[START CKPT(T1,T2)]
[T2,C,15]
[START T3]
[T1,D,20]
[COMMIT T1]
[T3,E,25]
[COMMIT T2]
[END CKPT]
[T3,F,30]

[START T1]
[T1,A,5]
[START T2]
[T2,B,10]
[START CKPT(T1,T2)]
[T2,C,15]
[START T3]
[T1,D,20]
[COMMIT T1]
[T3,E,25]
[COMMIT T2]
[END CKPT]
[T3,F,30]

Exercise 2 (Cntd)

- Undo/No-Redo Log
- T3 is the only uncommitted trx and must be undone.
- The recovery manager sets F to 30 and E to 25.

TOC
Recovery From System Failures

- Transactions
- Logging to recover from system failure
- Undo/Redo logging
- Undo logging
- Redo logging
  - The rules
  - The recovery procedure
  - Checkpointing
  - Exercise
- Finale

Redo Logging

The rules

Redo/No-Undo Logging

- Undo-work is needed because updates of uncommitted trx may have reached disk.
- Redo/No-Undo logging requires that no update of a trx reach the database on disk before the trx has committed.
- Advantage: avoids undo-work, and hence the need for before-images.
- WRITE(X,v) is logged as [T,X,⟨after_image⟩].
- Redo logging = Redo/No-Undo logging.

Rule for Redo Logging (1)

- *Rule R1:* 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,⟨after_image⟩] and the [COMMIT T] record, must appear on disk.

Rule for Redo Logging (2)

Since COMMIT record follows all update log records, the order in which material associated with one trx gets written to disk is:

1. The log records indicating changed database elements.
2. The COMMIT log record.
3. The changed database elements themselves.

Redo Logging

The recovery procedure

Recovery With Redo Logging

Basically like in Undo/Redo logging with Undo removed:

- Identify the committed trx.
- Scan the log forward from the beginning. For each record [T,X,⟨after_image⟩] encountered:
  - If T is not a committed trx, do nothing.
  - If T is committed, change the value of X in the database to ⟨after_image⟩.

Redo Logging

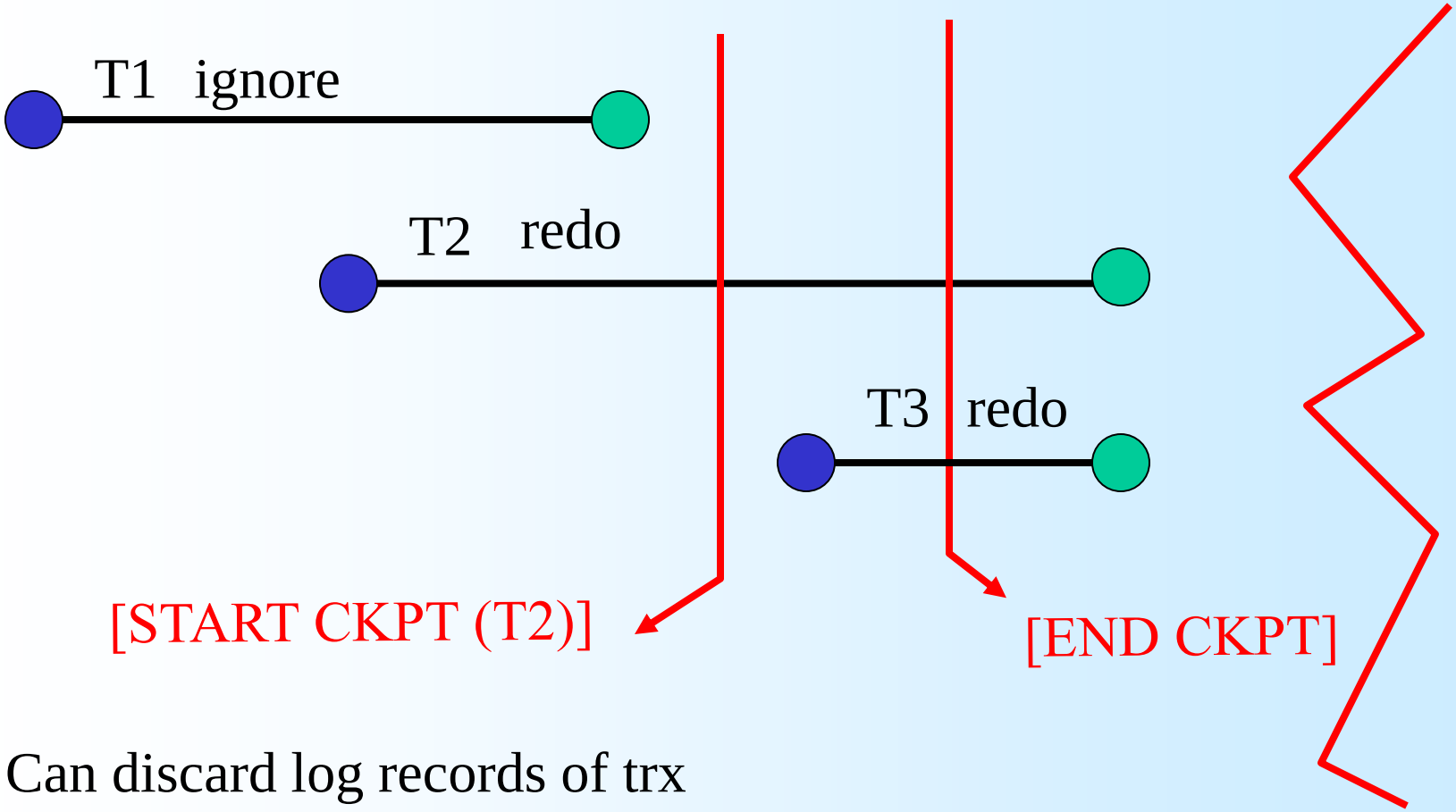
Checkpointing

Checkpointing a Redo Log

1. Write a [START CKPT (T_1,…,T_k)] record to the log, where T_1,…,T_k are the active trx, and flush the log.
2. Write to disk all database elements that were written to buffers but not yet to disk by transactions that had already committed when the START CKPT record was written to the log.
3. Write an [END CKPT] record to the log and flush the log.

Unlike Undo/Redo logging, where all dirty buffers are flushed.

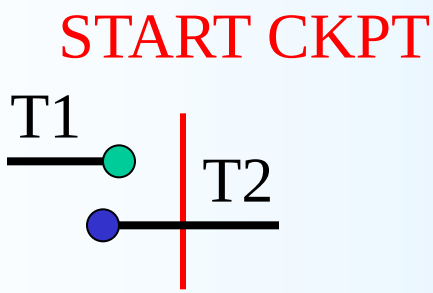
Checkpointing a Redo Log



Can discard log records of trx
that committed before START CKPT.

Disadvantages of Redo Logging

- Redo logging requires us to keep all modified blocks in buffers until the trx commits and the log records have been flushed, perhaps increasing the average number of buffers required by trx.
- Problems if database elements can share disk blocks.



T1's updates must reach disk, T2's updates must not.

Contradictory if T1 and T2 updated same memory block!

Redo Logging

Exercise

Exercise 3

- The following is the content of a **redo** log after a crash. Describe the changes to disk that have to be made by the recovery manager.

[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]

Exercise 3 (Cntd.)

[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]

In Redo/No-Undo logging, there is no need to undo the changes of the uncommitted trx T3.

All updates of T1 have reached the database.

T2 has to be redone.

- The recovery manager rewrites 10 for B and 15 for C.
- Note: We do not flush B to disk during the checkpoint.

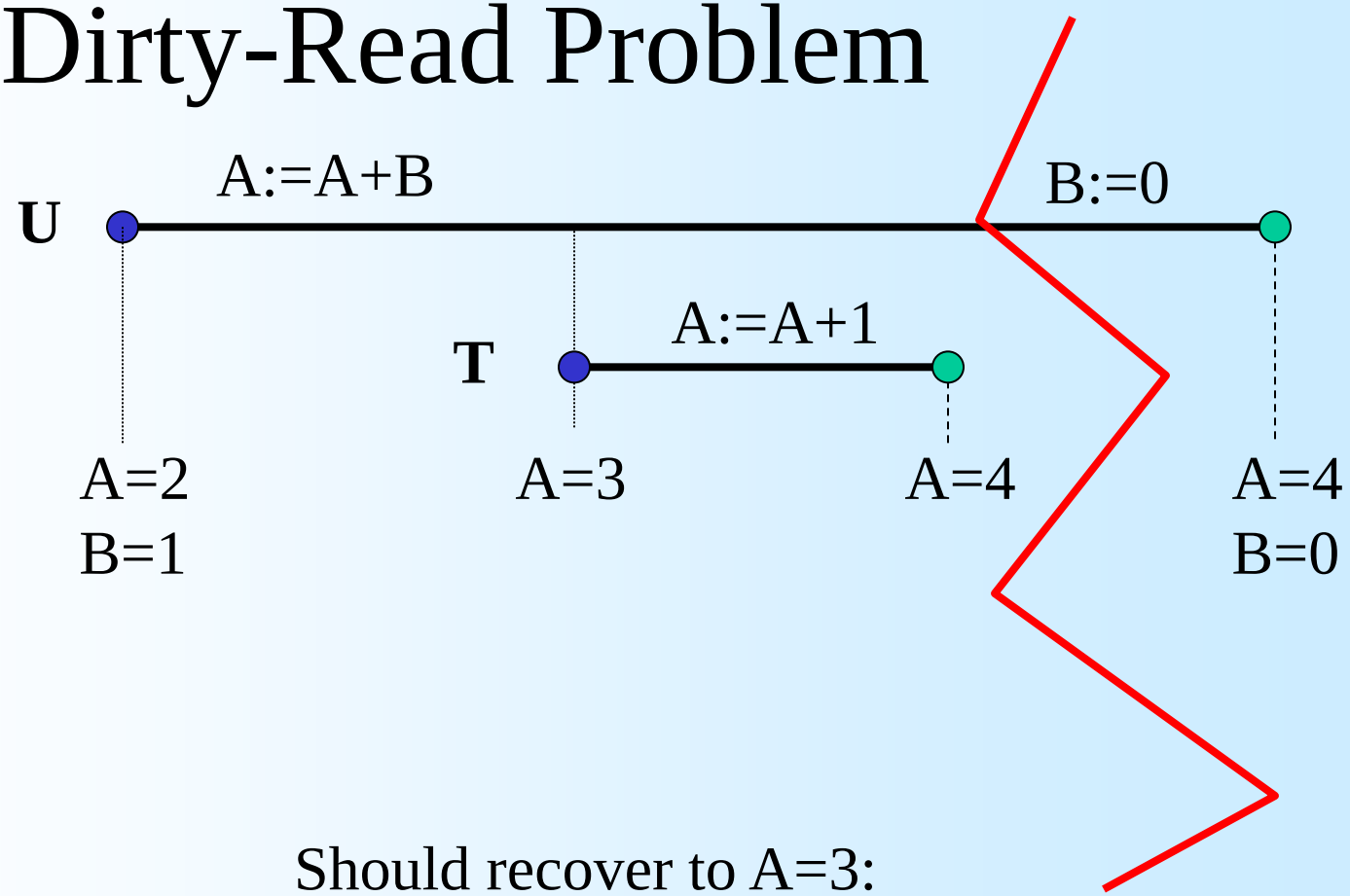
TOC
Recovery From System Failures

- Transactions
- Logging to recover from system failure
- Undo/Redo logging
- Undo logging
- Redo logging
- Finale

Summary

- Commit = END + FLUSH LOG
- Undo/Redo Logging:
  UR1: WRITE(A,v) < FLUSH LOG < OUTPUT(A)
  No constraint regarding Commit!
- Undo Logging:
  U1 = rule *UR1*
  U2: OUTPUT < Commit
- Redo Logging:
  R1: Commit < OUTPUT
  R1 implies *UR1* (as END is last operation of trx)

Dirty-Read Problem



[START U]
[U,A,2,3]
[START T]
[T,A,3,4]
[COMMIT T]
[U,B,1,0]
[COMMIT U]

Should recover to A=3:
starting from A=2, execute T, but not U.

However, undoing U yields A=2,
redoing T yields A=4.

Individual Trx Failure

- System failure erases main-memory, and hence all trx in progress. Certain conditions (e.g., division by zero) may cause an individual trx T to fail.
- If trx T fails, its updates that reached the database must be undone. (like with system failure)
- Also, updates of T that reside in the buffer must be canceled! (unlike system failure)
- An [ABORT T] record will be written in the log to indicate that T could not complete successfully and has been rolled back.

Recovery From System Failures
END

COMING NEXT:
Concurrency Control