

# Database System Internals

## Transactions: Recovery (part 2)

Paul G. Allen School of Computer Science and Engineering  
University of Washington, Seattle

# Announcements

- Lab 3 due Tuesday evening
- Lab 4 out tonight, start reading the spec!

# 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

Easy to implement (how?) and ensures atomicity

# No-Force/Steal (most 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

Aries: is an UNDO-REDO log

# Policies and Logs

	<b>NO-STEAL</b>	<b>STEAL</b>
<b>FORCE</b>	Lab 3	Undo Log
<b>NO-FORCE</b>	Redo Log	Undo-Redo Log

# "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



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>

WHAT DO WE DO ?

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	 Crash !
COMMIT						<COMMIT T>

WHAT DO WE DO ?

We **UNDO** by setting B=8 and A=8

# After Crash

- This is all we see (for example):

Disk A	Disk B
8	16

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

# After Crash

- This is all we see (for example):

Disk A	Disk B
8	16

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

# After Crash

- This is all we see (for example):
- Need to step through the log

Disk A	Disk B
8	16

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

# After Crash

- This is all we see (for example):
- Need to step through the log

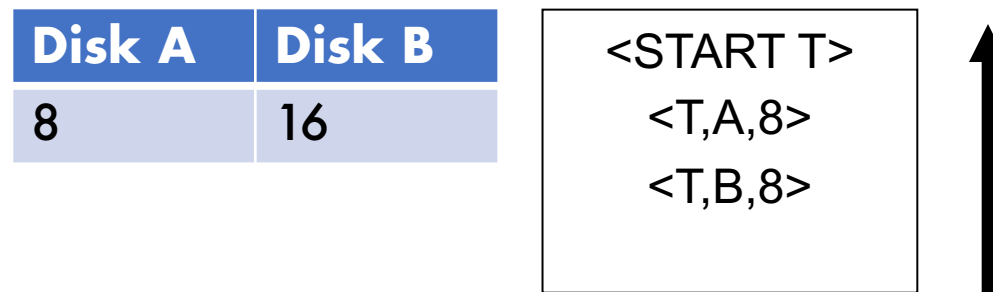
Disk A	Disk B
8	16

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

- What direction?

# After Crash

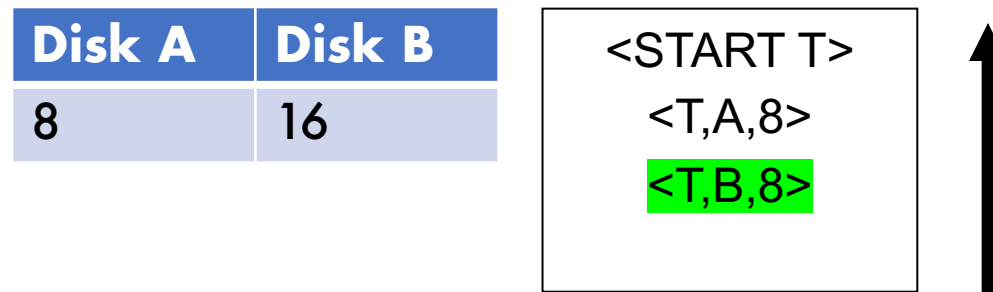
- This is all we see (for example):
- Need to step through the log



- What direction?
- In UNDO log, we start at the most recent and go backwards in time

# After Crash

- This is all we see (for example):
- Need to step through the log

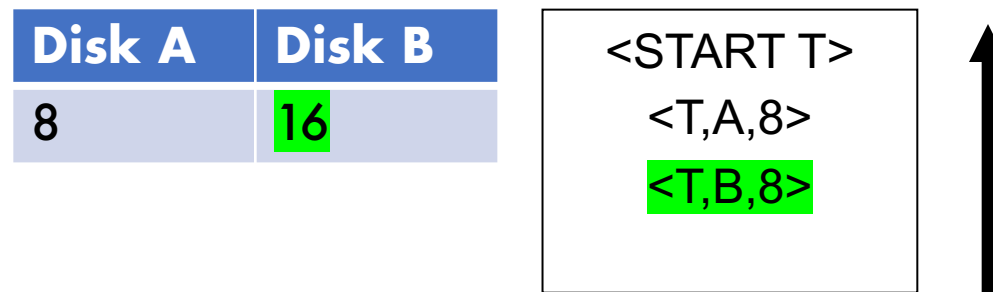


- What direction?
- In UNDO log, we start at the most recent and go backwards in time



# After Crash

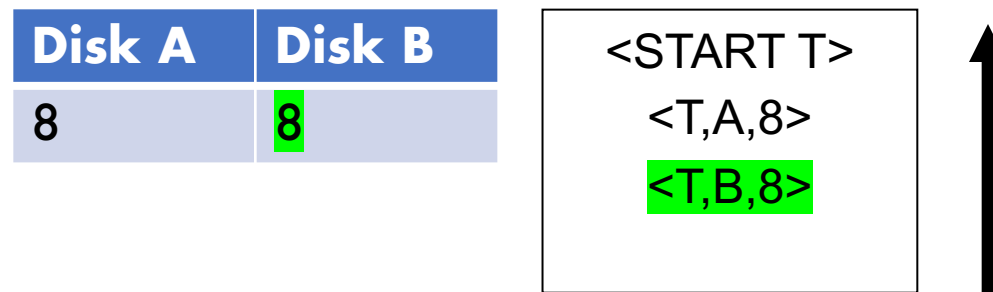
- This is all we see (for example):
- Need to step through the log



- What direction?
- In UNDO log, we start at the most recent and go backwards in time

# After Crash

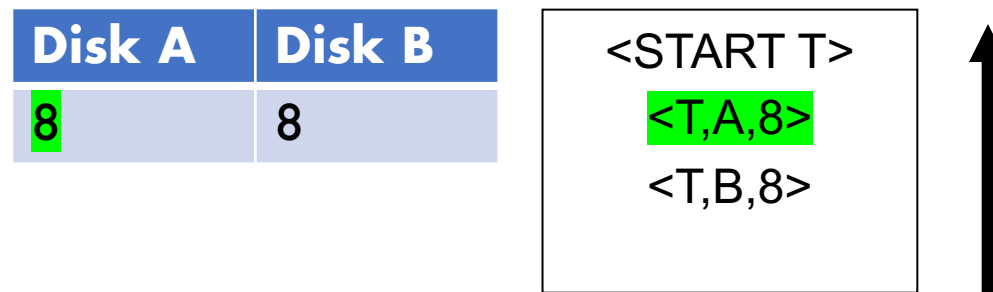
- This is all we see (for example):
- Need to step through the log



- What direction?
- In UNDO log, we start at the most recent and go backwards in time

# After Crash

- This is all we see (for example):
- Need to step through the log



- What direction?
- In UNDO log, we start at the most recent and go backwards in time

# After Crash

- 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
- In 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 with Undo Log

After system's crash, run recovery manager

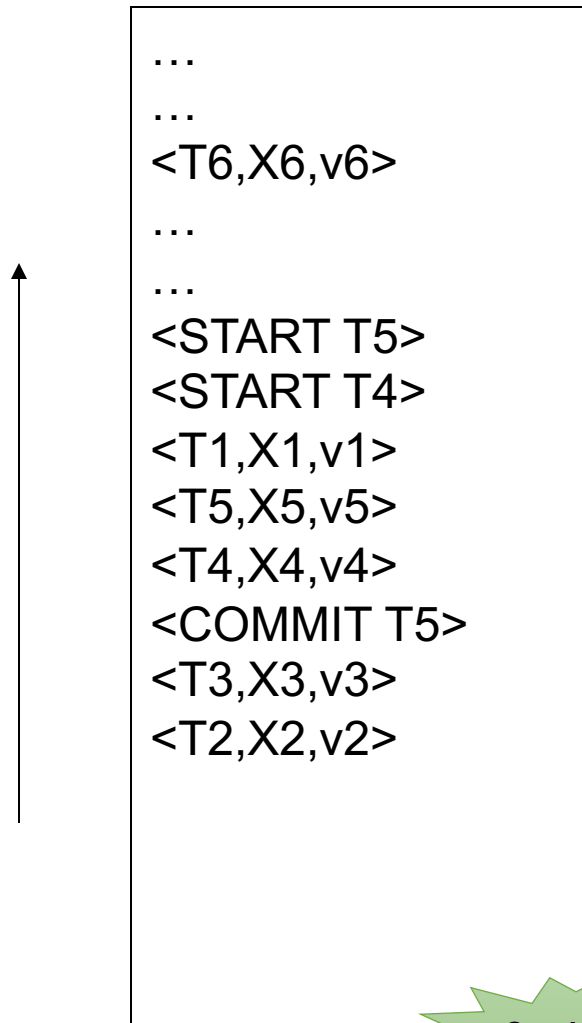
- Decide for each transaction T whether it is completed or not
  - $\langle \text{START } T \rangle \dots \langle \text{COMMIT } T \rangle \dots$  = yes
  - $\langle \text{START } T \rangle \dots \langle \text{ABORT } T \rangle \dots$  = yes (already cleaned up)
  - $\langle \text{START } T \rangle \dots$  = no
- Undo all modifications by **incomplete** transactions

# Recovery with Undo Log

Recovery manager:

- Read log from the end; 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

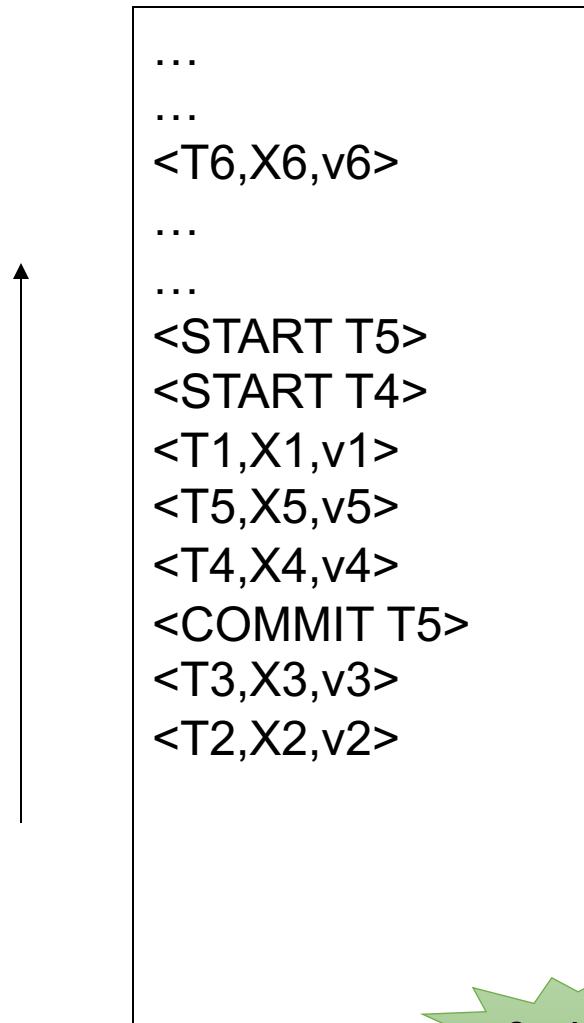


**Question 1:** Which updates are undone ?

**Question 2:**  
How far back do we need to read in the log ?

**Question 3:**  
What happens if second crash during recovery?

# Recovery with Undo Log



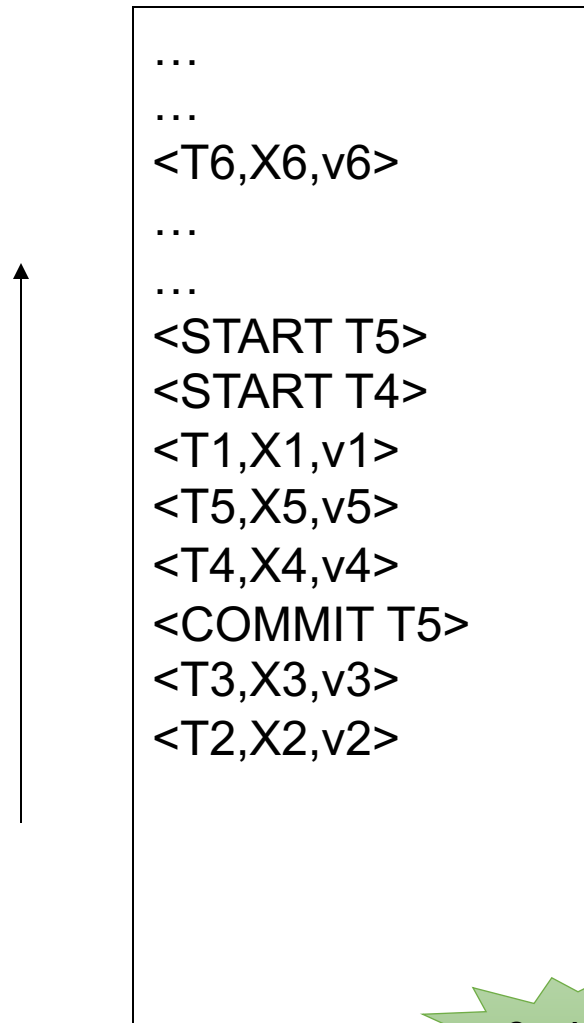
**Question 1:** Which updates are undone ?

**Question 2:**  
How far back do we need to read in the log ?  
*To the beginning.*

**Question 3:**  
What happens if second crash during recovery?



# Recovery with Undo Log



**Question1:** Which updates are undone ?

**Question 2:**  
How far back do we need to read in the log ?  
To the beginning.

**Question 3:**  
What happens if second crash during recovery?  
No problem! Log records are idempotent. Can reapply.

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO Log
						<START T>
INPUT(A)					8	
READ(A,t)	8				8	
t:=t*2	16	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>

When must we force pages to disk ?



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>

RULES: log entry before OUTPUT before COMMIT

# Undo-Logging Rules

U1: If T modifies X, then  $\langle T, X, v \rangle$  must be written to disk before  $\text{OUTPUT}(X)$

U2: If T commits, then  $\text{OUTPUT}(X)$  must be written to disk before  $\langle \text{COMMIT } T \rangle$

- Hence: OUTPUTs are done early, before the transaction commits



FORCE

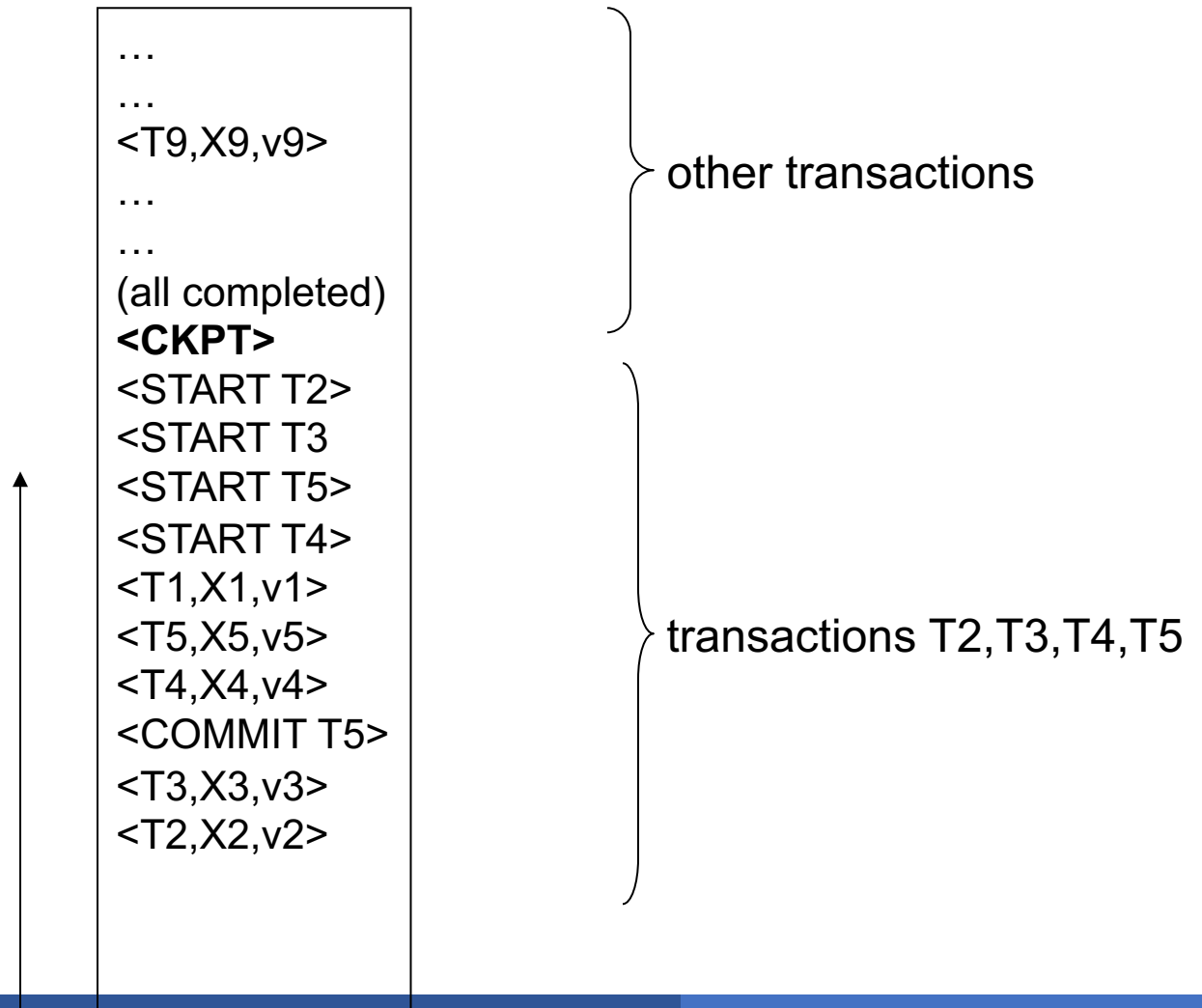
# Checkpointing

Checkpoint the database periodically

- **Stop** accepting new transactions
- Wait until all current transactions complete
- Flush log to disk
- Write a <CKPT> log record, flush
- **Resume** transactions

# Undo Recovery with Checkpointing

During recovery,  
Can stop at first  
<CKPT>



# Nonquiescent Checkpointing

- Problem with checkpointing: database freezes during checkpoint
- Would like to checkpoint while database is operational
- Idea: nonquiescent checkpointing

Quiescent = being quiet, still, or at rest; inactive  
Non-quiescent = allowing transactions to be active

# Nonquiescent Checkpointing

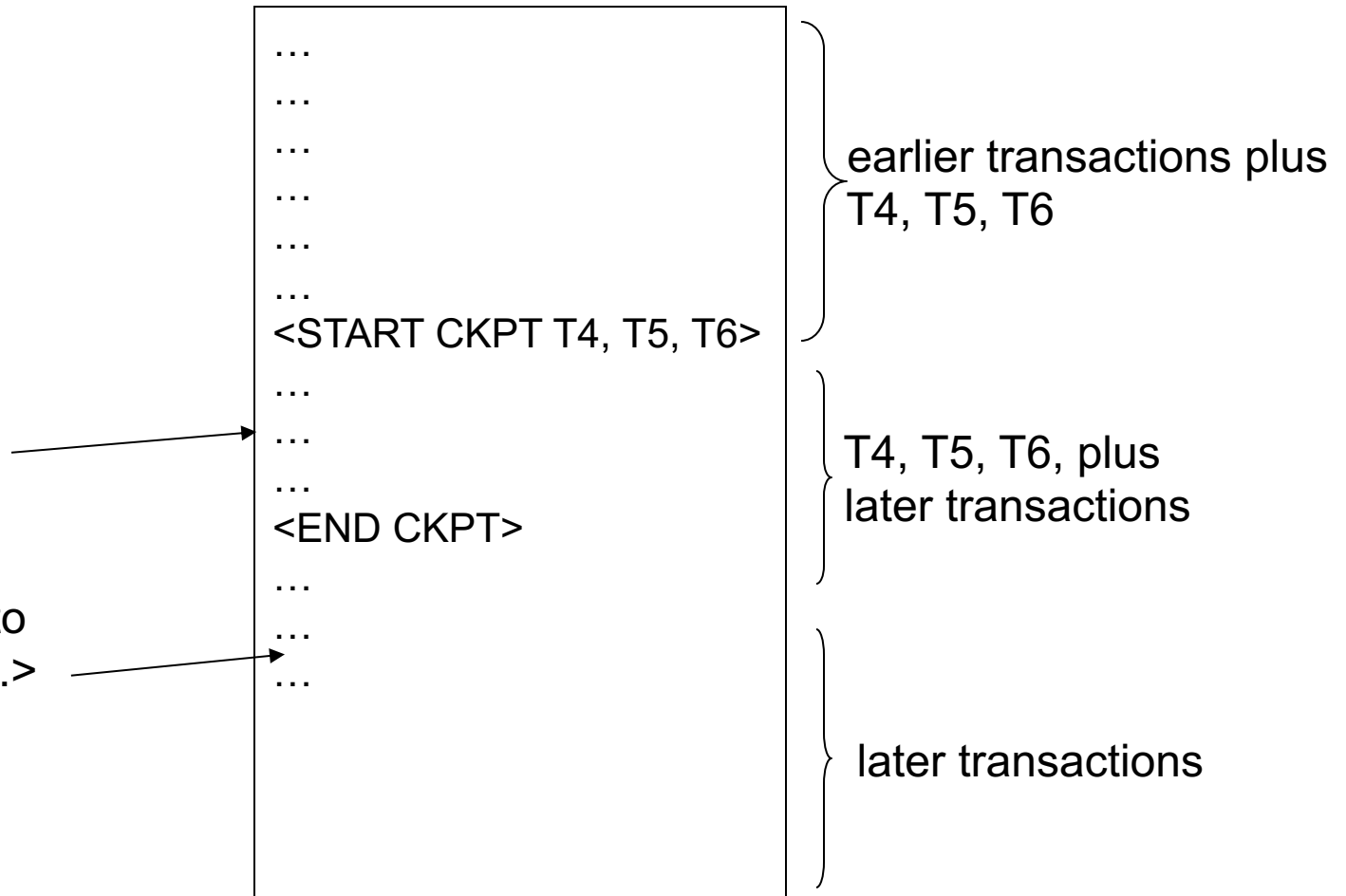
- Write a  $\langle \text{START CKPT}(T_1, \dots, T_k) \rangle$  where  $T_1, \dots, T_k$  are all active transactions. Flush log to disk
- Continue normal operation
- When all of  $T_1, \dots, T_k$  have **completed**, write  $\langle \text{END CKPT} \rangle$ , flush log to disk



# Undo with Nonquiescent Checkpointing

If we crash here:  
Need to read  
Back to start of  
T4, T5, T6

If we crash here:  
Need to read only to  
<START CKPT T4..>



# Implementing ROLLBACK

- Recall: a transaction can end in COMMIT or ROLLBACK
- Idea: use the undo-log to implement ROLLBACK
- How ?
  - LSN = Log Sequence Number
  - Log entries for the same transaction are linked, using the LSN's
  - Read log in reverse, using LSN pointers

# Implementing ROLLBACK

- Record
- RO

- Idea

- How

- 
- 
- 

...

...

<T9,X9,v9>

...

...

(all completed)

**<CKPT>**

<START T2>

<START T3>

<START T5>

<START T4>

<T1,X1,v1>

<T5,X5,v5>

<T2,X1,v2>

<T4,X4,v4>

<COMMIT T5>

<T3,X3,v3>

<T2,X2,v2>

CK

sing



# REDO

NO-FORCE and NO-STEAL

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		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

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		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

Crash !

Is this bad ?

Yes, it's bad: A=16, B=8

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		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

Crash !

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		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

Crash !



Is this bad ?

Yes, it's bad: lost update

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		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

Crash !

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		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

Crash !

Is this bad ?

No: that's OK.

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		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

Crash !

# Redo Logging

One minor change to the undo log:

- $\langle T, X, v \rangle =$  T has updated element X, and its new value is v

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
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,16>
COMMIT						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
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,16>
COMMIT						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Crash !

How do we recover ?

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
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,16>
COMMIT						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Crash !

How do we recover ?

We **REDO** by setting A=16 and B=16

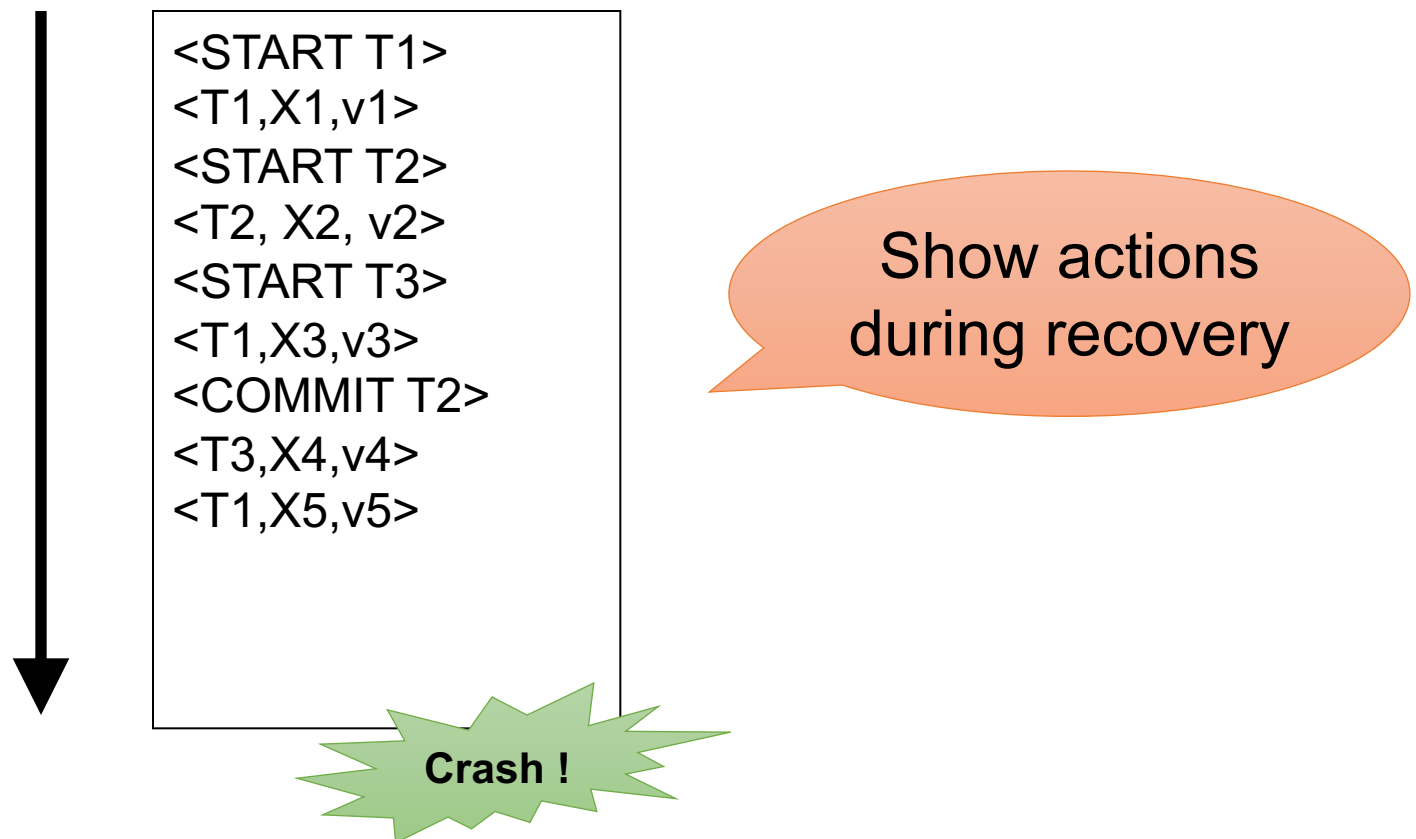
# Recovery with Redo Log

After system's crash, run recovery manager

- Step 1. Decide for each transaction T whether it is committed or not
  - <START T>....<COMMIT T>.... = yes
  - <START T>....<ABORT T>..... = no
  - <START T>..... = no
- Step 2. Read log from the beginning, redo all updates of committed transactions



# Recovery with Redo Log



# Nonquiescent Checkpointing

- Write a  $\langle \text{START CKPT}(T_1, \dots, T_k) \rangle$   
where  $T_1, \dots, T_k$  are all active txn's
- Begin flush to disk all **blocks of committed transactions** (*dirty blocks*)
- Meantime, continue normal operation
- When **all blocks have been written**, write  $\langle \text{END CKPT} \rangle$

**END CKPT has different meaning here than in Undo log!**  
It does not mean that  $T_1, \dots, T_k$  are complete

# Nonquiescent Checkpointing

Step 1: look for  
The last  
<END CKPT> and it's  
<START CKPT>

All OUTPUTs  
of T1 are  
known to be on disk

Cannot  
use

```
...  
<START T1>  
...  
<COMMIT T1>  
...  
<START T4>  
...  
<START CKPT T4, T5, T6>  
...  
...  
...  
...  
<END CKPT>  
...  
...  
<START CKPT T9, T10>  
...
```

Step 2: redo  
from the  
earliest  
start of  
T4, T5, T6  
ignoring  
transactions  
committed  
earlier

Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8			8	
t:=t*2	16	8			8	
WRITE(A,t)	16	16		8	8	<T,A,16>
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,16>
COMMIT						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

When must we force pages to disk ?



Action	t	Mem A	Mem B	Disk A	Disk B	REDO Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
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,16>
COMMIT						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

RULE: OUTPUT after COMMIT

NO-STEAL

# Redo-Logging Rules

R1: If T modifies X, then both  $\langle T, X, v \rangle$  and  $\langle \text{COMMIT } T \rangle$  must be written to disk before  $\text{OUTPUT}(X)$

- Hence: OUTPUTs are done late

**NO-STEAL**

# Comparison Undo/Redo

## ▪ Undo logging:

Steal/Force

- 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

No-Steal/No-Force

- 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 (next)

Steal/No-Force

# Undo/Redo Logging

Log records, only one change

- $\langle T, X, u, v \rangle$  = T has updated element X, its old value was u, and its new value is v



# Undo/Redo-Logging Rule

UR1: If  $T$  modifies  $X$ , then  $\langle T, X, u, v \rangle$  must be written to disk before  $\text{OUTPUT}(X)$

Note: we are free to  $\text{OUTPUT}$  early or late relative to  $\langle \text{COMMIT } T \rangle$

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8,16>
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,8,16>
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

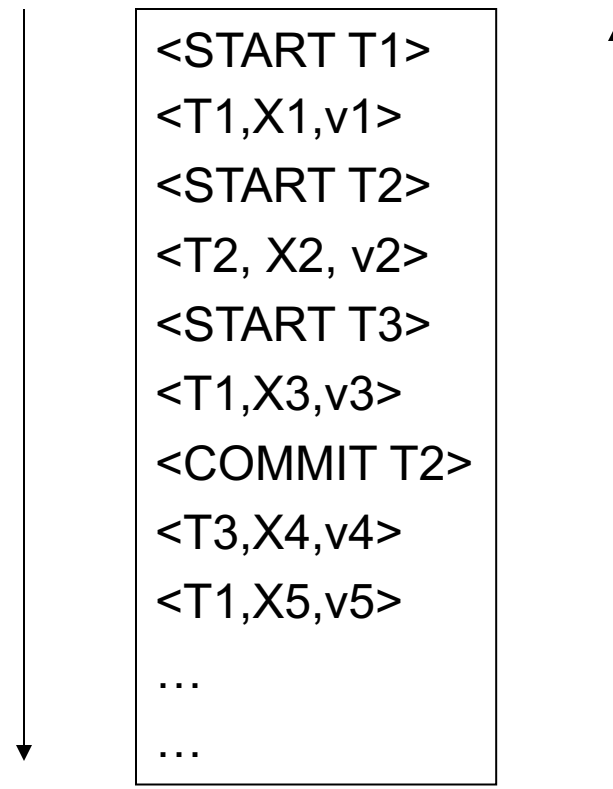
Can OUTPUT whenever we want: before/after COMMIT

# Recovery with Undo/Redo Log

After system's crash, run recovery manager

- Redo all committed transaction, top-down
- Undo all uncommitted transactions, bottom-up

# Recovery with Undo/Redo Log



# ARIES

# Aries

- ARIES pieces together several techniques into a comprehensive algorithm
- Developed at IBM Almaden, by Mohan
- IBM botched the patent, so everyone uses it now
- Several variations, e.g. for distributed transactions

# Log Granularity

Two basic types of log records for update operations

- **Physical log records**

- Position on a particular page where update occurred
- Both before and after image for undo/redo logs
- Benefits: Idempotent & updates are fast to redo/undo

- **Logical log records**

- Record only high-level information about the operation
- Benefit: Smaller log
- BUT difficult to implement because crashes can occur in the middle of an operation

# ARIES Recovery Manager

Log entries:

- **<START T>** – when T begins
- **Update: <T,X,u,v>**
  - T updates X, old value=u, new value=v
  - Logical description of the change
- **<COMMIT T>** or **<ABORT T>** then **<END>**
- **<CLR>** – we'll talk about them later.



# ARIES Recovery Manager

Rule:

- If T modifies X, then  $\langle T, X, u, v \rangle$  must be written to disk before  $\text{OUTPUT}(X)$

We are free to  $\text{OUTPUT}$  early or late w.r.t commits

# LSN = Log Sequence Number

- **LSN** = identifier of a log entry
  - Log entries belonging to the same TXN are linked with extra entry for previous LSN
- Each page contains a **pageLSN**:
  - LSN of log record for latest update to that page

# ARIES Data Structures

## ▪ Active Transactions Table

- Lists all active TXN's
- For each TXN: **lastLSN** = its most recent update LSN

## ▪ Dirty Page Table

- Lists all dirty pages
- For each dirty page: **recoveryLSN** (**recLSN**) = first LSN that caused page to become dirty

## ▪ Write Ahead Log

- LSN, **prevLSN** = previous LSN for same txn

$W_{T100}(P7)$  $W_{T200}(P5)$  $W_{T200}(P6)$  $W_{T100}(P5)$ 

# Data Structures

## Dirty pages

pageID	recLSN
P5	102
P6	103
P7	101

## Log (WAL)

LSN	prevLSN	transID	pageID	Log entry
101	-	T100	P7	
102	-	T200	P5	
103	102	T200	P6	
104	101	T100	P5	

## Active transactions

transID	lastLSN
T100	104
T200	103

## Buffer Pool

P8	P2	...
	...	
P5 PageLSN=104	P6 PageLSN=103	P7 PageLSN=101

# ARIES Normal Operation

T writes page P

- What do we do ?

# ARIES Normal Operation

T writes page P

▪ What do we do ?

- Write  $\langle T, P, u, v \rangle$  in the Log
- $\text{pageLSN} = \text{LSN}$
- $\text{prevLSN} = \text{lastLSN}$
- $\text{lastLSN} = \text{LSN}$
- $\text{recLSN} = \text{if isNull then LSN}$

# ARIES Normal Operation

Buffer manager wants to OUTPUT(P)

- What do we do ?

Buffer manager wants INPUT(P)

- What do we do ?

# ARIES Normal Operation

Buffer manager wants to OUTPUT(P)

- Flush log up to **pageLSN**
- Remove P from **Dirty Pages** table

Buffer manager wants INPUT(P)

- What do we do ?



# ARIES Normal Operation

Buffer manager wants to OUTPUT(P)

- Flush log up to **pageLSN**
- Remove P from **Dirty Pages** table

Buffer manager wants INPUT(P)

- Create entry in **Dirty Pages** table  
**recLSN** = NULL

# ARIES Normal Operation

Transaction T starts

- What do we do ?

Transaction T commits/aborts

- What do we do ?

# ARIES Normal Operation

Transaction T starts

- Write **<START T>** in the log
- New entry T in Active TXN;  
lastLSN = null

Transaction T commits

- What do we do ?

# ARIES Normal Operation

Transaction T starts

- Write **<START T>** in the log
- New entry T in Active TXN;  
lastLSN = null

Transaction T commits

- Write **<COMMIT T>** in the log
- Flush log up to this entry
- Write **<END>**

# Checkpoints

Write into the log

- Entire **active transactions table**
- Entire **dirty pages table**

Recovery always starts by analyzing latest checkpoint

Background process periodically flushes dirty pages to disk

# ARIES Recovery

## 1. Analysis pass

- Figure out what was going on at time of crash
- List of dirty pages and active transactions

## 2. Redo pass (repeating history principle)

- Redo all operations, even for transactions that will not commit
- Get back to state at the moment of the crash

## 3. Undo pass

- Remove effects of all uncommitted transactions
- Log changes during undo in case of another crash during undo

# 1. Analysis Phase

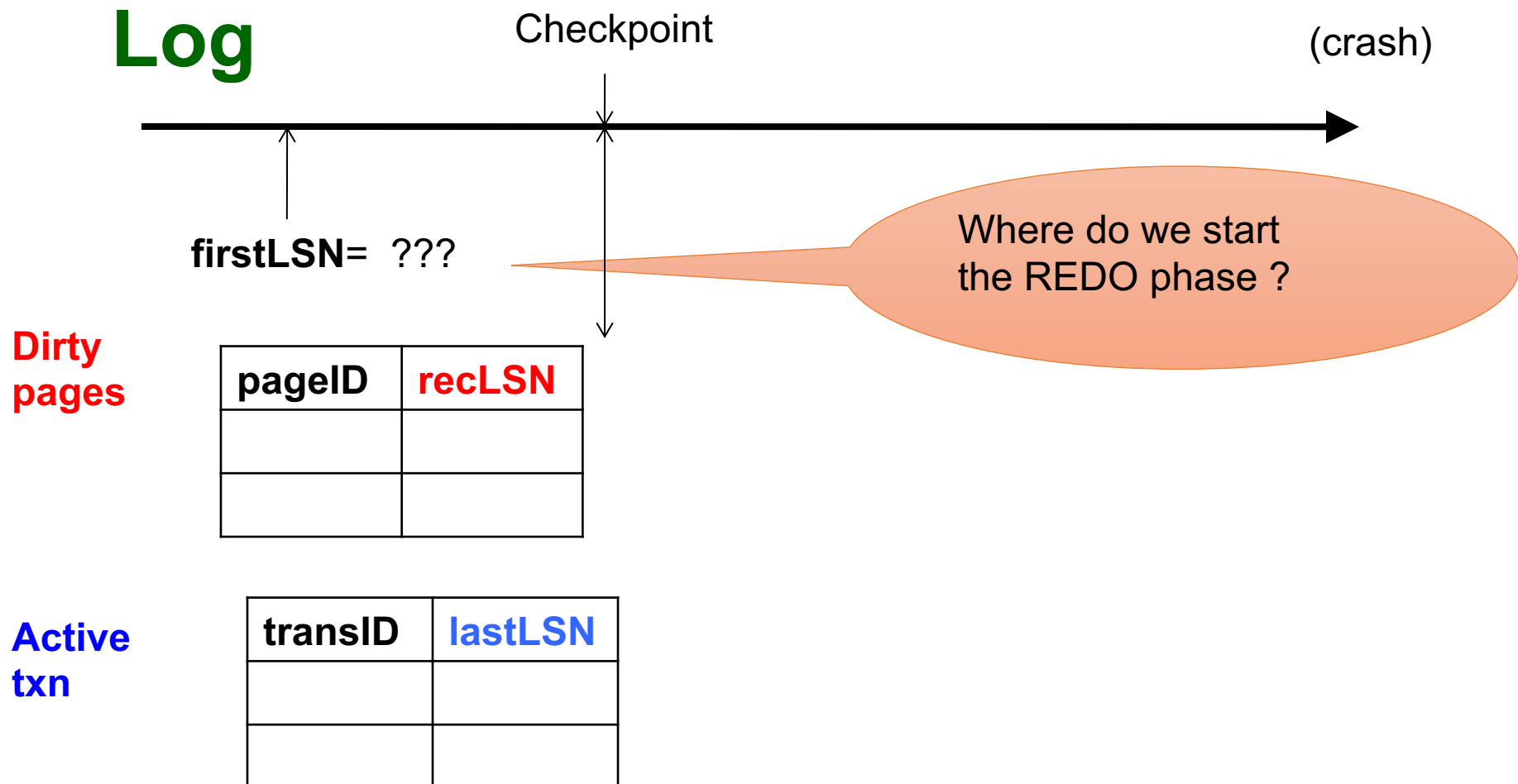
## ■ Goal

- Determine point in log where to start REDO
- Determine set of dirty pages when crashed
  - Conservative estimate of dirty pages
- Identify active transactions when crashed

## ■ Approach

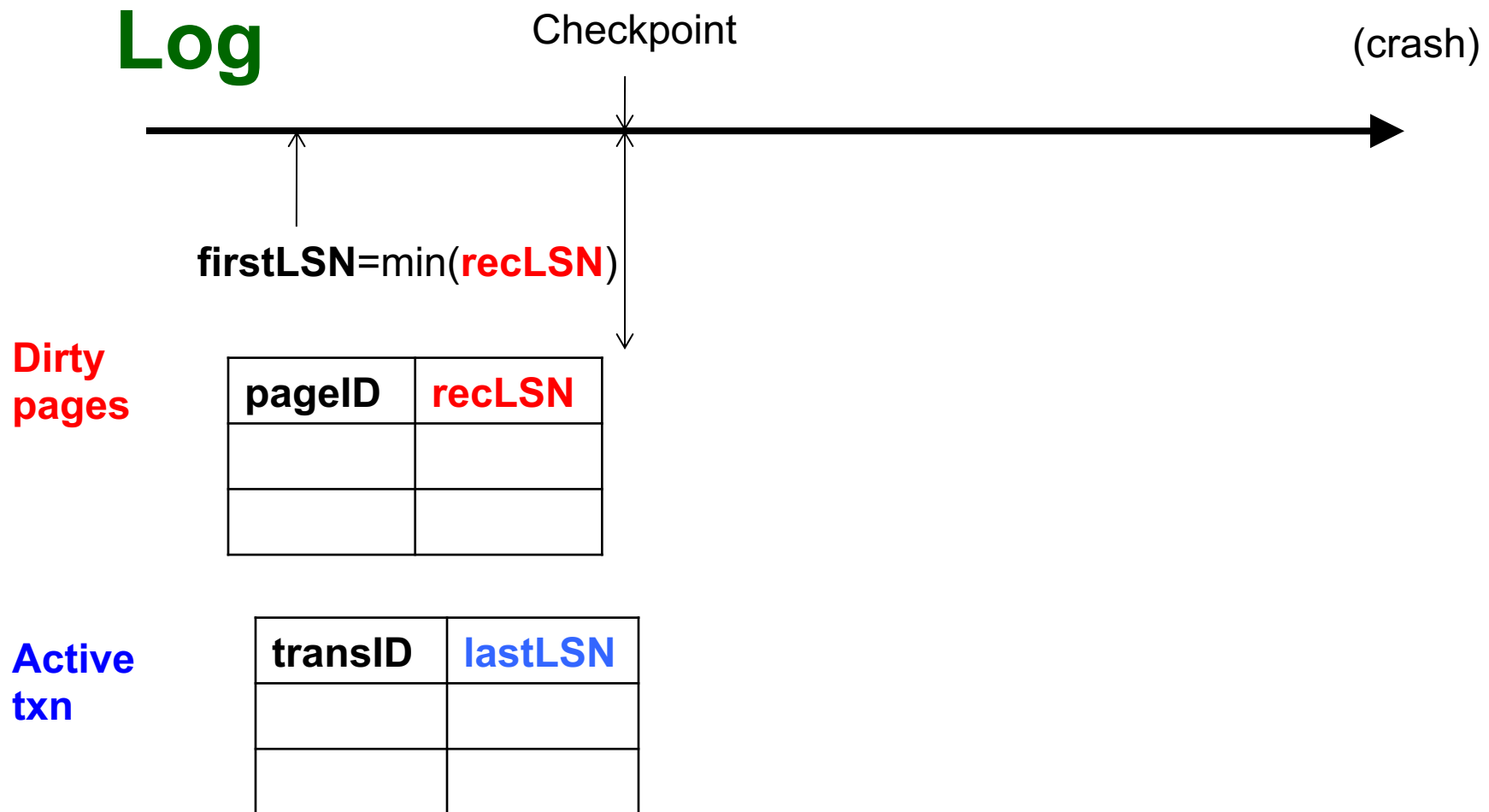
- Rebuild **active transactions table** and **dirty pages table**
- Reprocess the log from the checkpoint
  - Only update the two data structures
- Compute: **firstLSN** = smallest of all **recoveryLSN**

# 1. Analysis Phase

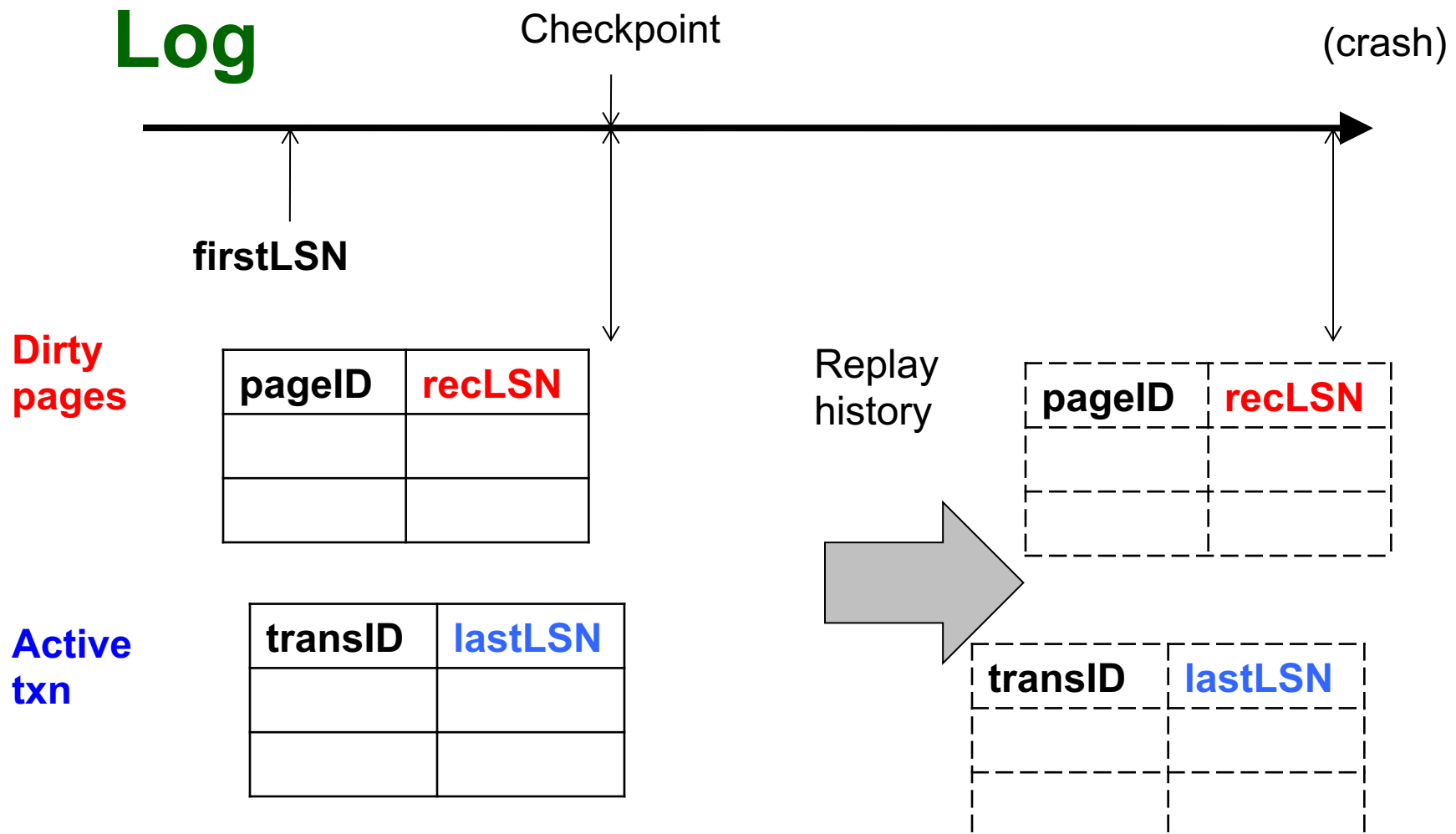




# 1. Analysis Phase



# 1. Analysis Phase



## 2. Redo Phase

Main principle: replay history

- Process Log forward, starting from **firstLSN**
- Read every log record, sequentially
- Redo actions are not recorded in the log
- Needs the **Dirty Page Table**

## 2. Redo Phase: Details

For each **Log** entry record **LSN:  $\langle T, P, u, v \rangle$**

- Redo the action  $P=u$  and  $WRITE(P)$
- Only redo actions that need to be redone

## 2. Redo Phase: Details

For each **Log** entry record **LSN:  $\langle T, P, u, v \rangle$**

- If **P** is not in **Dirty Page** then **no update**
- If **recLSN** > **LSN**, then **no update**
- Read page from disk:  
If **pageLSN** >= **LSN**, then **no update**
- Otherwise perform update

## 2. Redo Phase: Details

What happens if system crashes during REDO ?

## 2. Redo Phase: Details

What happens if system crashes during REDO ?

We REDO again ! The pageLSN will ensure that we do not reapply a change twice

### 3. Undo Phase

- Cannot “unplay” history, in the same way as we “replay” history
- WHY NOT ?



### 3. Undo Phase

- Cannot “unplay” history, in the same way as we “replay” history
- WHY NOT ?
  - Undo only the loser transactions
  - Need to support ROLLBACK: selective undo, for one transaction
- Hence, *logical* undo v.s. *physical* redo

### 3. Undo Phase

Main principle: “logical” undo

- Start from end of **Log**, move backwards
- Read only affected log entries
- Undo actions *are* written in the Log as special entries: **CLR** (Compensating Log Records)
- **CLR**s are redone, but never undone

### 3. Undo Phase: Details

- “Loser transactions” = uncommitted transactions in **Active Transactions Table**
- **ToUndo** = set of **lastLSN** of loser transactions

### 3. Undo Phase: Details

While **ToUndo** not empty:

- Choose most recent (largest) **LSN** in **ToUndo**
- If **LSN** = regular record **<T,P,u,v>**:
  - Write a **CLR** where **CLR.undoNextLSN** = **LSN.prevLSN**
  - Undo **v**
- If **LSN** = **CLR** record:
  - Don't undo !
- if **CLR.undoNextLSN** not null, insert in **ToUndo** otherwise, write **<END>** in log

### 3. Undo Phase: Details

What happens if system crashes during UNDO ?

### 3. Undo Phase: Details

What happens if system crashes during UNDO ?

We do not UNDO again ! Instead, each CLR is a REDO record: we simply redo the undo