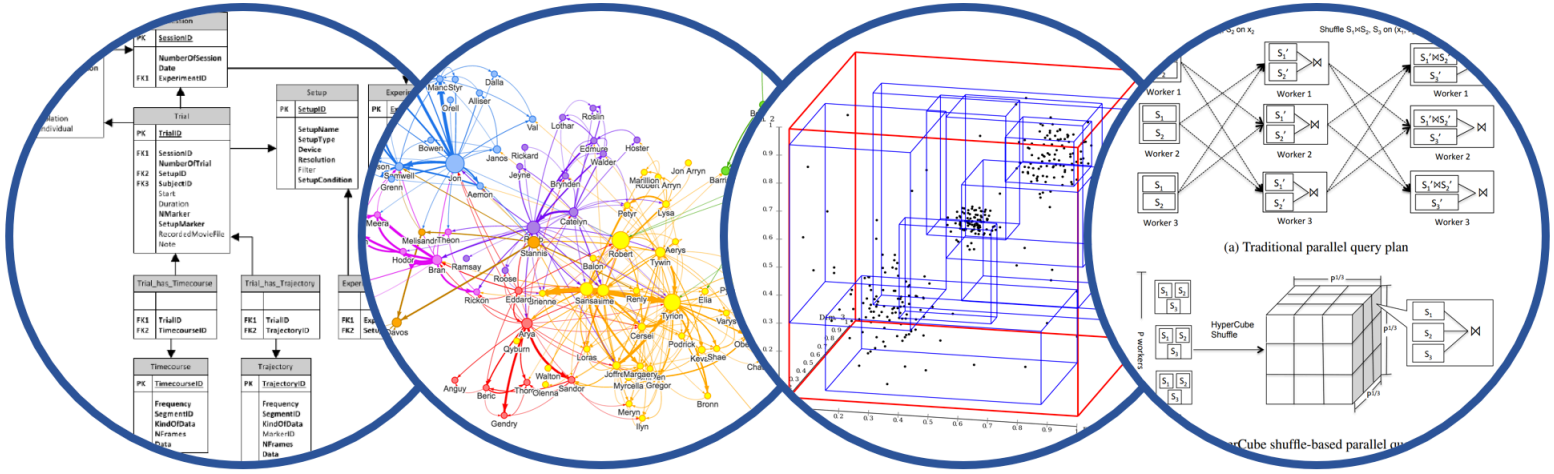


Database System Internals

Transactions: Recovery (part 3)

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

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 (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

Aries: is an UNDO-REDO log

Policies and Logs

	NO-STEAL	STEAL
FORCE	Lab 3	Undo Log
NO-FORCE	Redo Log	Undo-Redo Log

Action	t	Mem A	M…	…k 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 <T,X,v> and <COMMIT T> must be written to disk before 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

- <T,X,u,v>= 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 <T,X,u,v> must be written to disk before OUTPUT(X)

Note: we are free to OUTPUT early or late relative to <COMMIT T>

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
REAT(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	

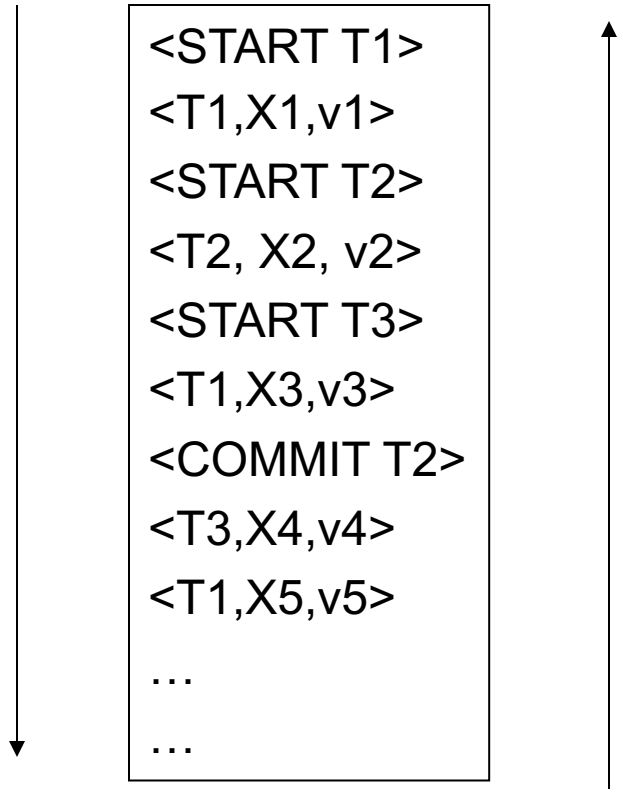
Can OUTPUT whenever we want: before/after COMMIT

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

Undo/Redo protocol

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

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 <T,X,u,v> must be written to disk before OUTPUT(X)

We are free to 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

Data Structures

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

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 **<T,P,u,v>** in the **Log**
- **pageLSN**=**LSN**
- **prevLSN**=**lastLSN**
- **lastLSN**=**LSN**
- **recLSN**=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

ARIES Method Illustration

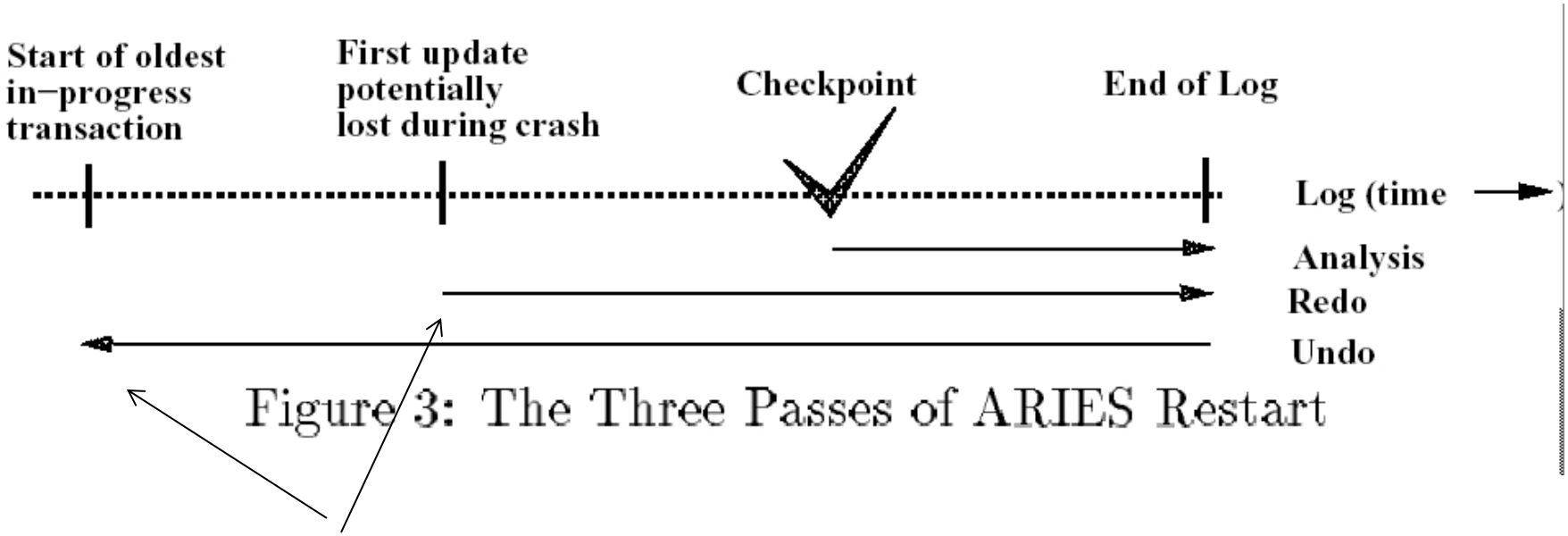


Figure 3: The Three Passes of ARIES Restart

First undo and first redo log entry might be in reverse order

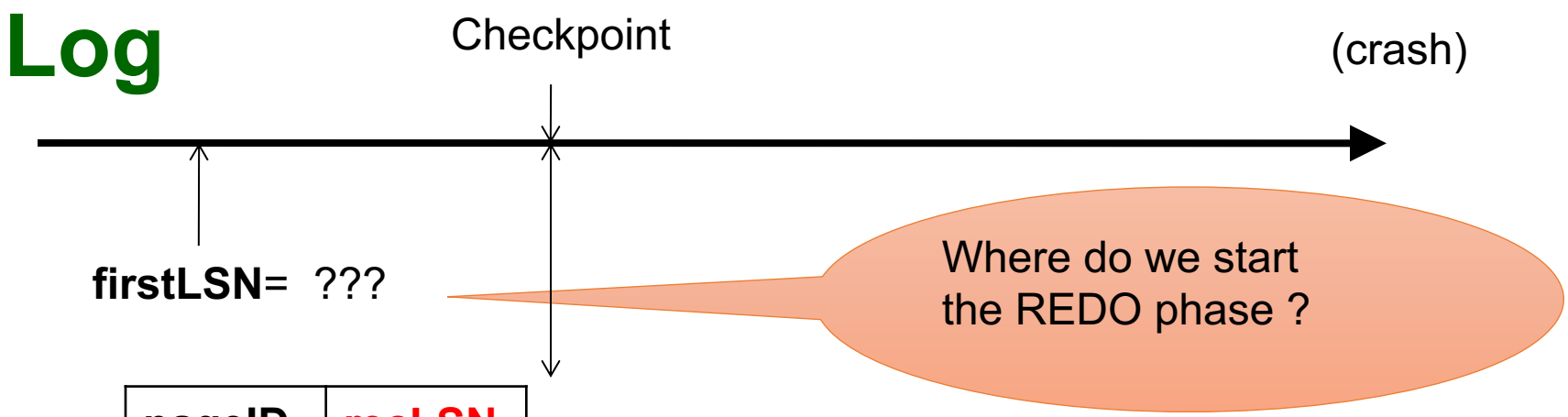
[Figure 3 from Franklin97]

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



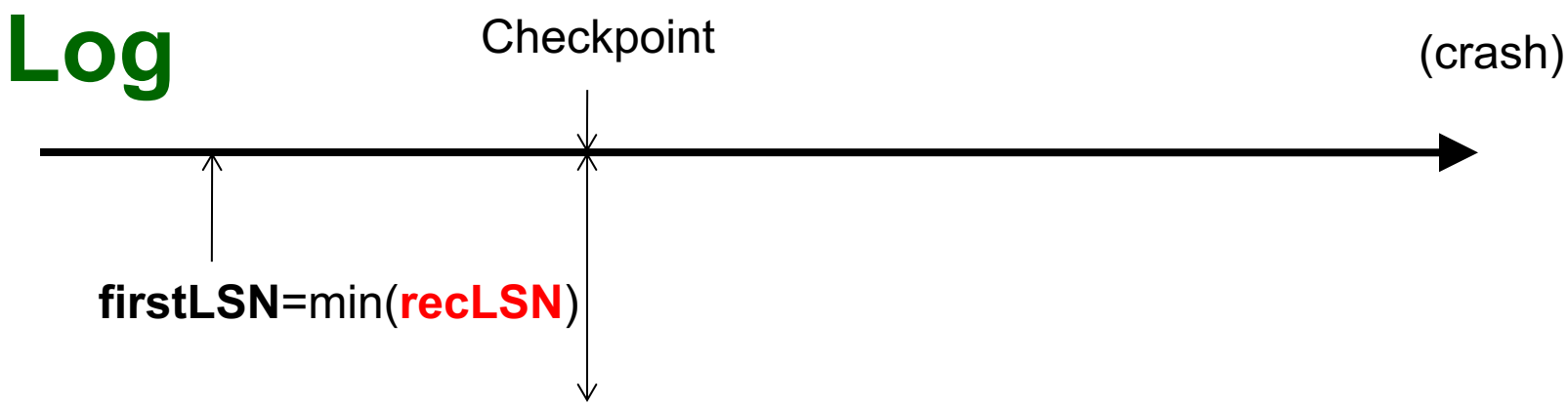
Dirty pages

pageID	recLSN

Active txn

transID	lastLSN

1. Analysis Phase



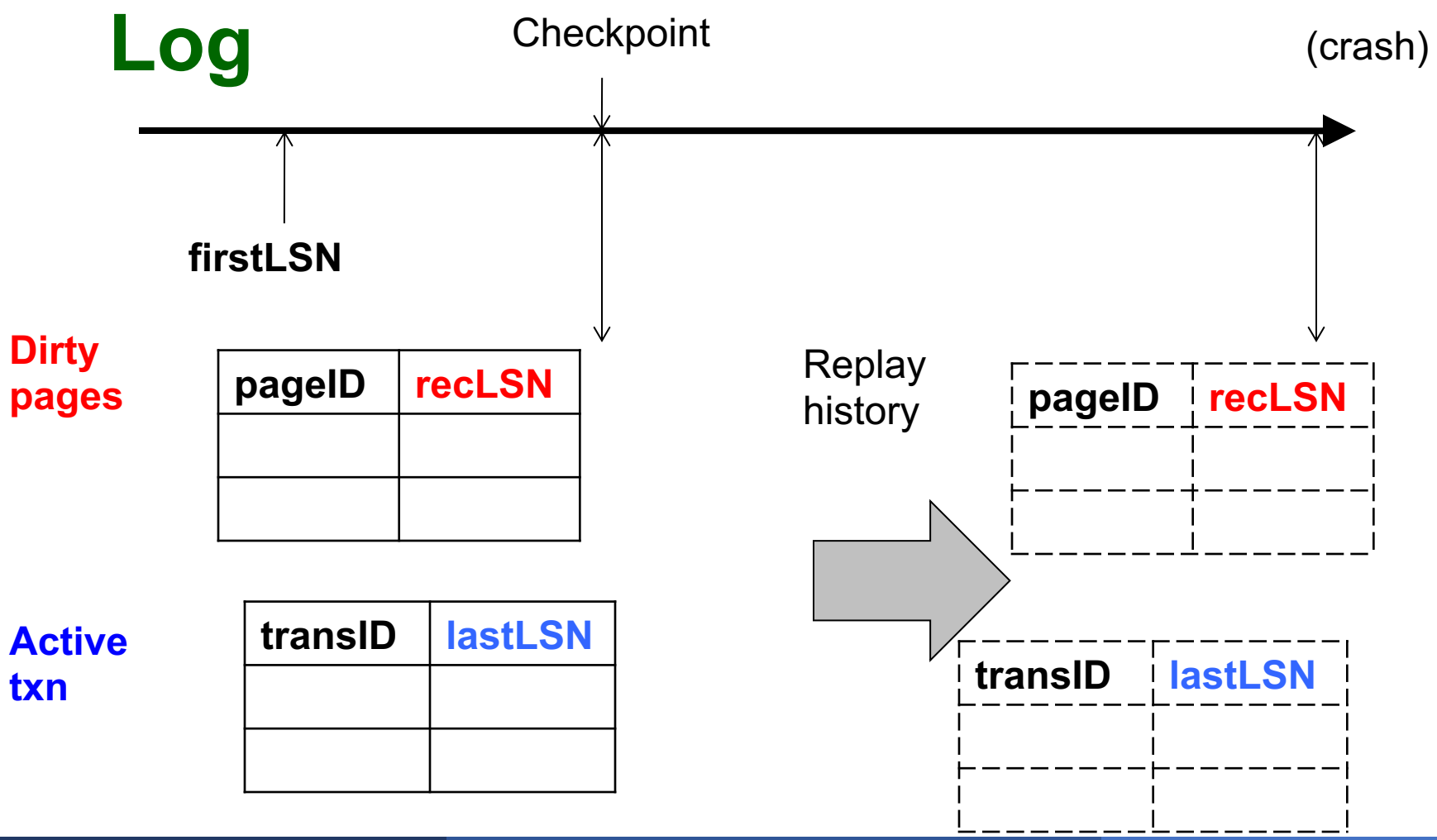
Dirty pages

pageID	recLSN

Active txn

transID	lastLSN

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: **<T,P,u,v>**

- 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: **<T,P,u,v>**

- 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 ?
  - We can only undo only the loser transactions
  - Need to support ROLLBACK: selective undo, for one transaction

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)
- CLRs 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

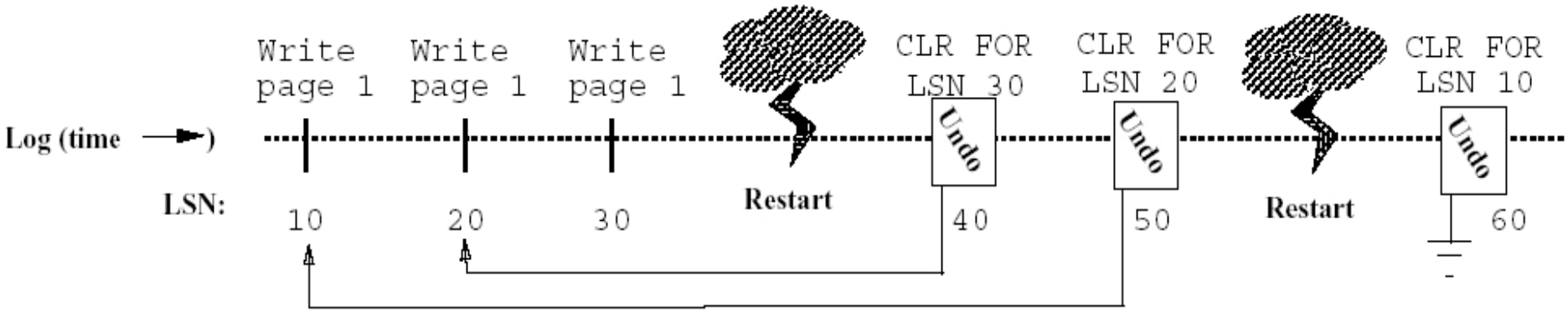


Figure 4: The Use of CLRs for UNDO

[Figure 4 from Franklin97]

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