Database Design and Implementation

CS 645

Recovery

Review: the ACID properties

- Atomicity
  - All actions of a Xact happen, or none happen
- Consistency
  - If each Xact is consistent, and the DB starts consistent, it ends up consistent
- Isolation
  - Execution of one Xact is isolated from others
- Durability
  - If a Xact commits, its effects persist

Which ones does the Recovery Manager help with?

*(also consistency related rollbacks)

Primitive Operations of Transactions

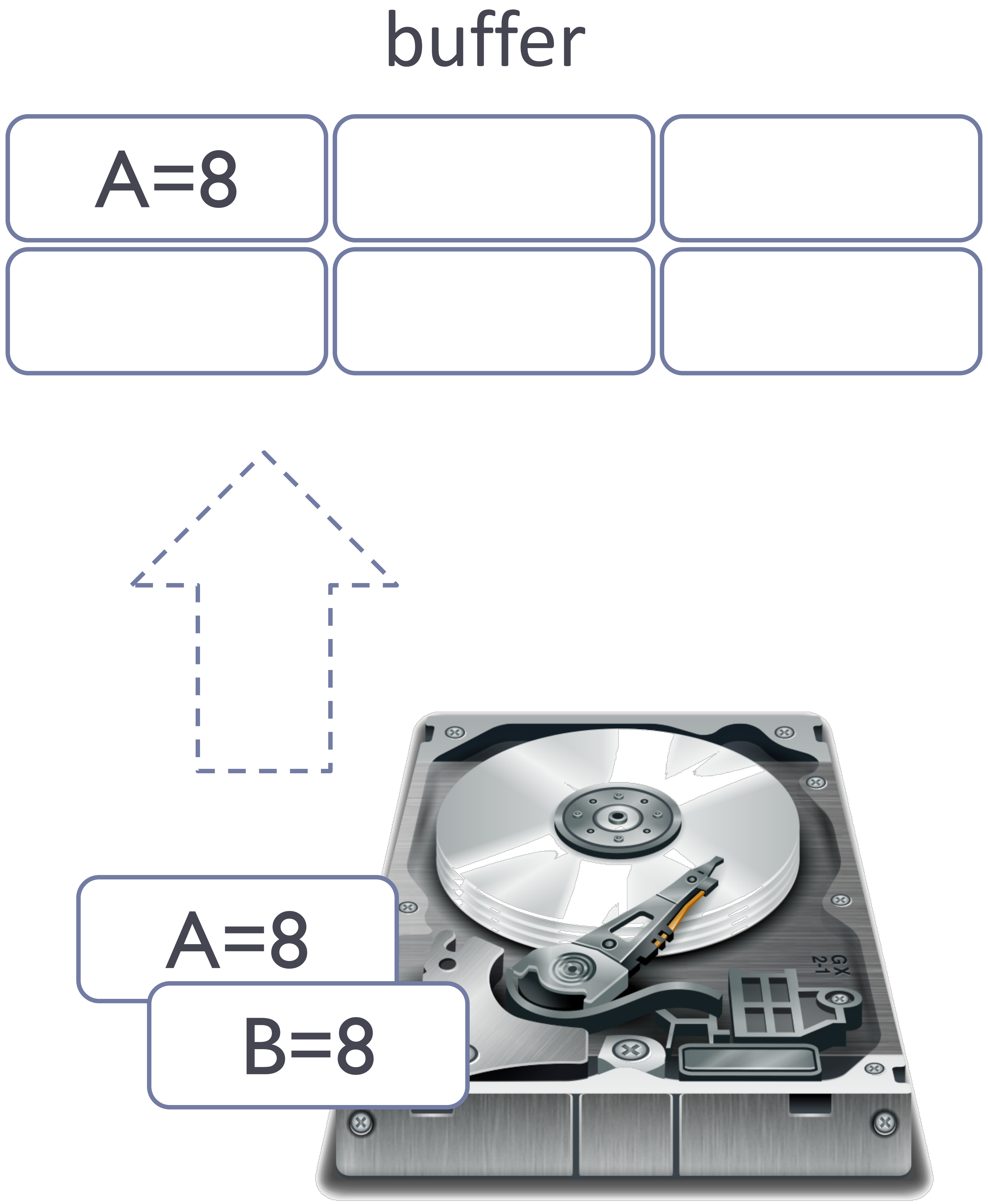
- READ(X,t)
  - copy element X to transaction local variable t
- WRITE(X,t)
  - copy transaction local variable t to element X

- INPUT(X)
  - read element X to memory buffer
- OUTPUT(X)
  - write element X to disk

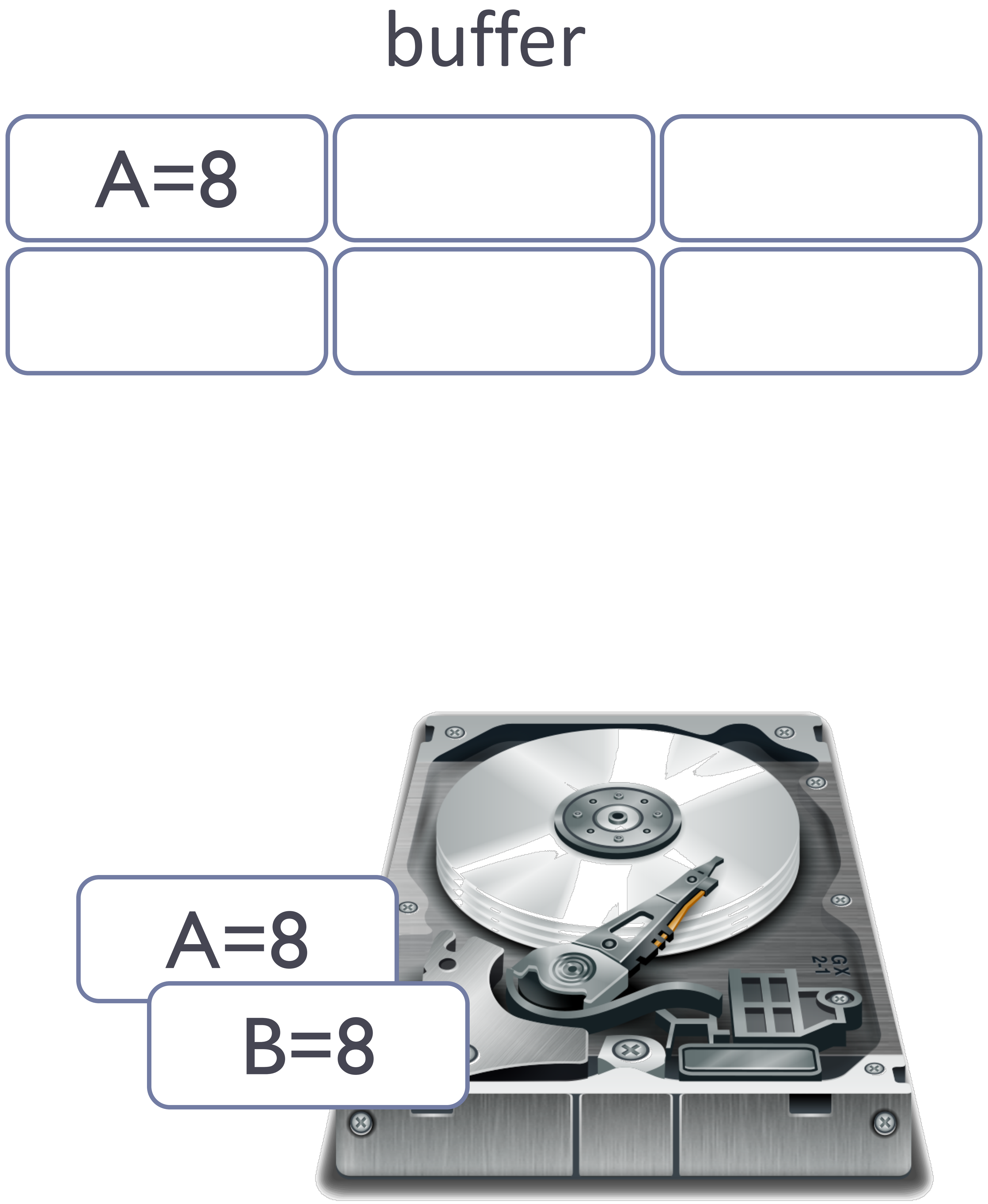
Example

```
START TRANSACTION
READ(A,t);
t := t*2;
WRITE(A,t);
READ(B,t);
t := t*2;
WRITE(B,t);
COMMIT;
```

Atomicity:
BOTH A and B
are multiplied by 2



```
START TRANSACTION
READ(A,t);
t := t*2;
WRITE(A,t);
READ(B,t);
t := t*2;
WRITE(B,t);
COMMIT;
```



buffer

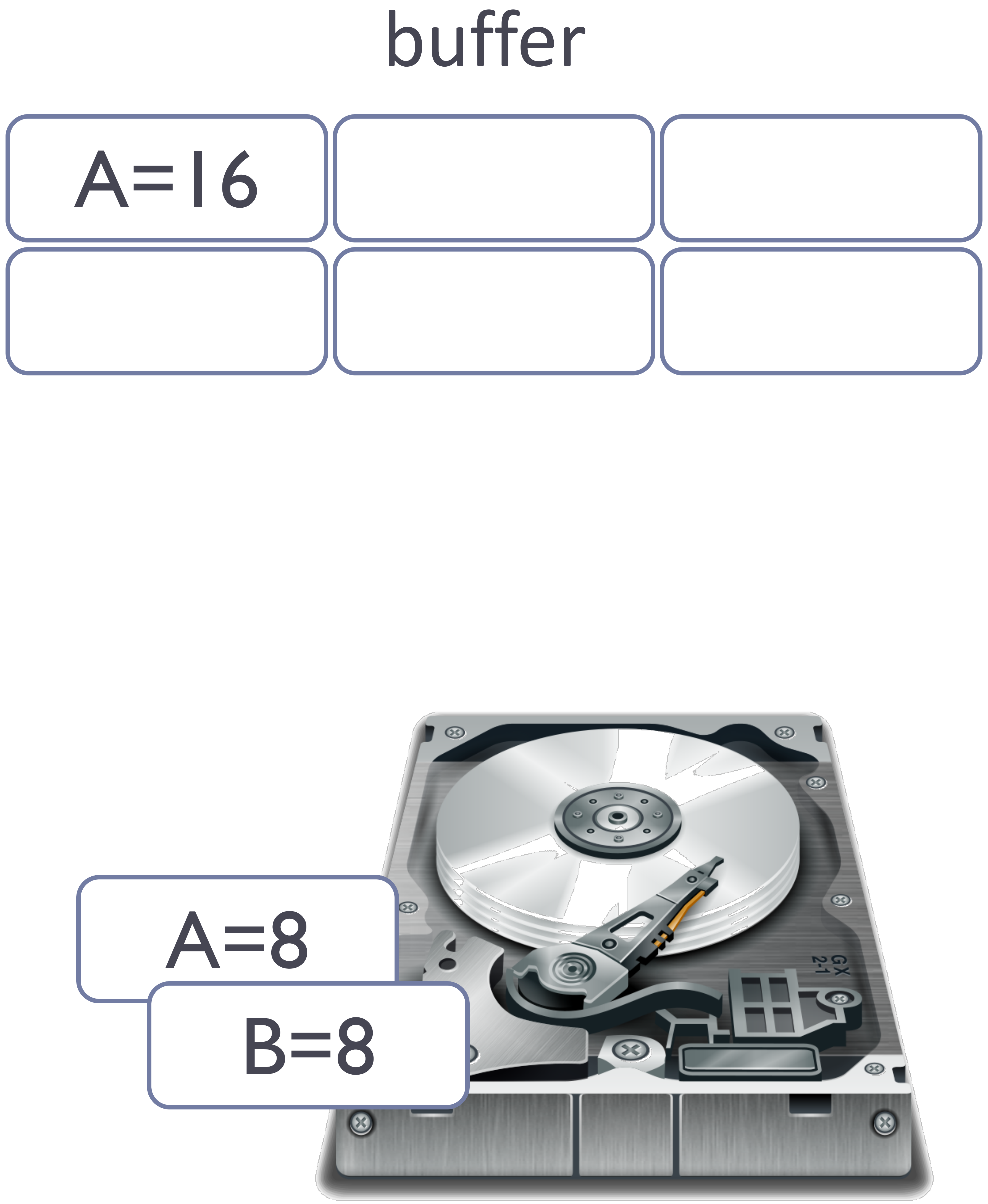
A=8

A=8

B=8

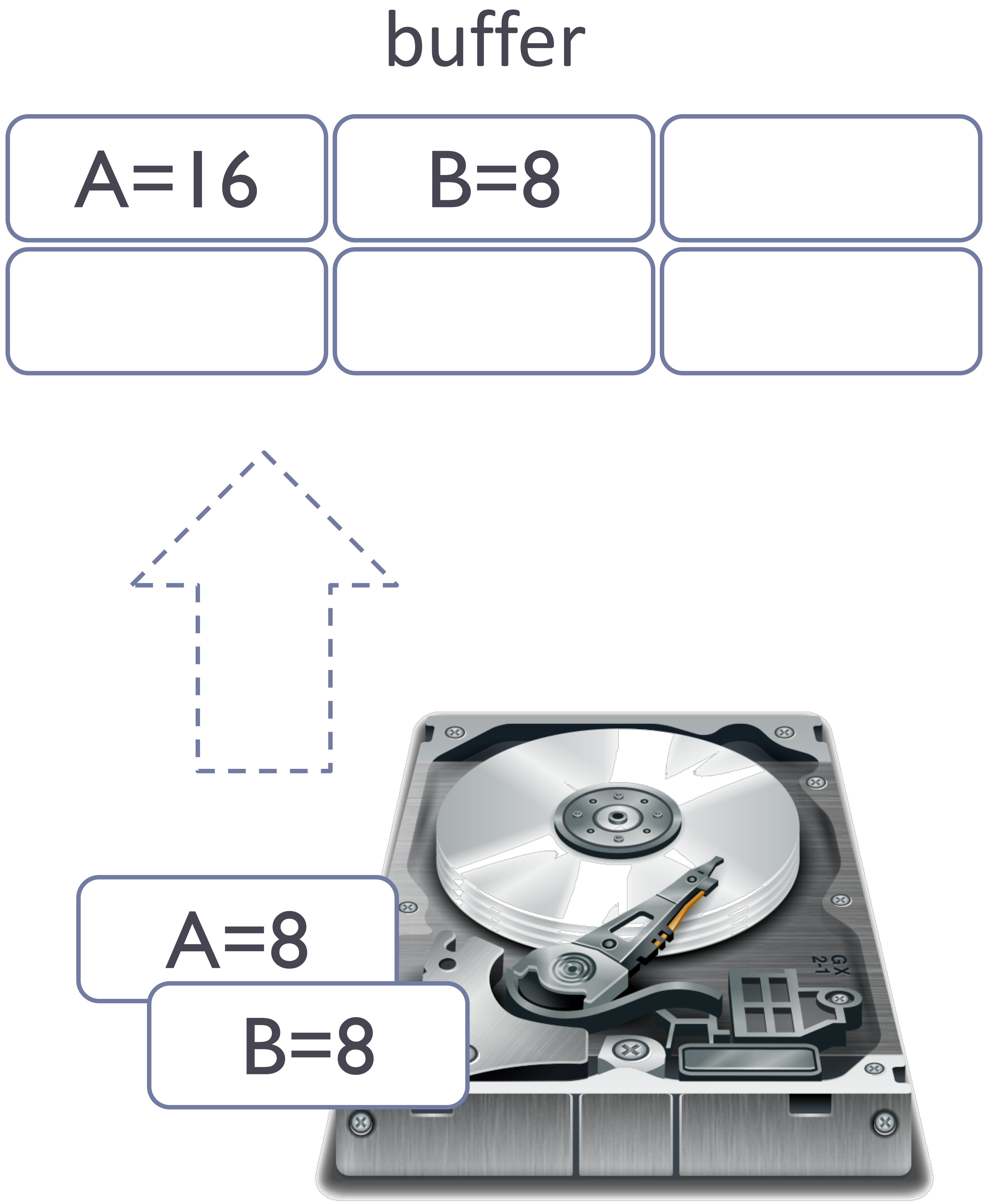
```
START TRANSACTION
READ(A,t);
t := t*2;
WRITE(A,t);
READ(B,t);
t := t*2;
WRITE(B,t);
COMMIT;
```

t=8



```
START TRANSACTION
READ(A,t);
t := t*2;
WRITE(A,t);
READ(B,t);
t := t*2;
WRITE(B,t);
COMMIT;
```

t=16



buffer

A=16	B=8	

A=8

B=8

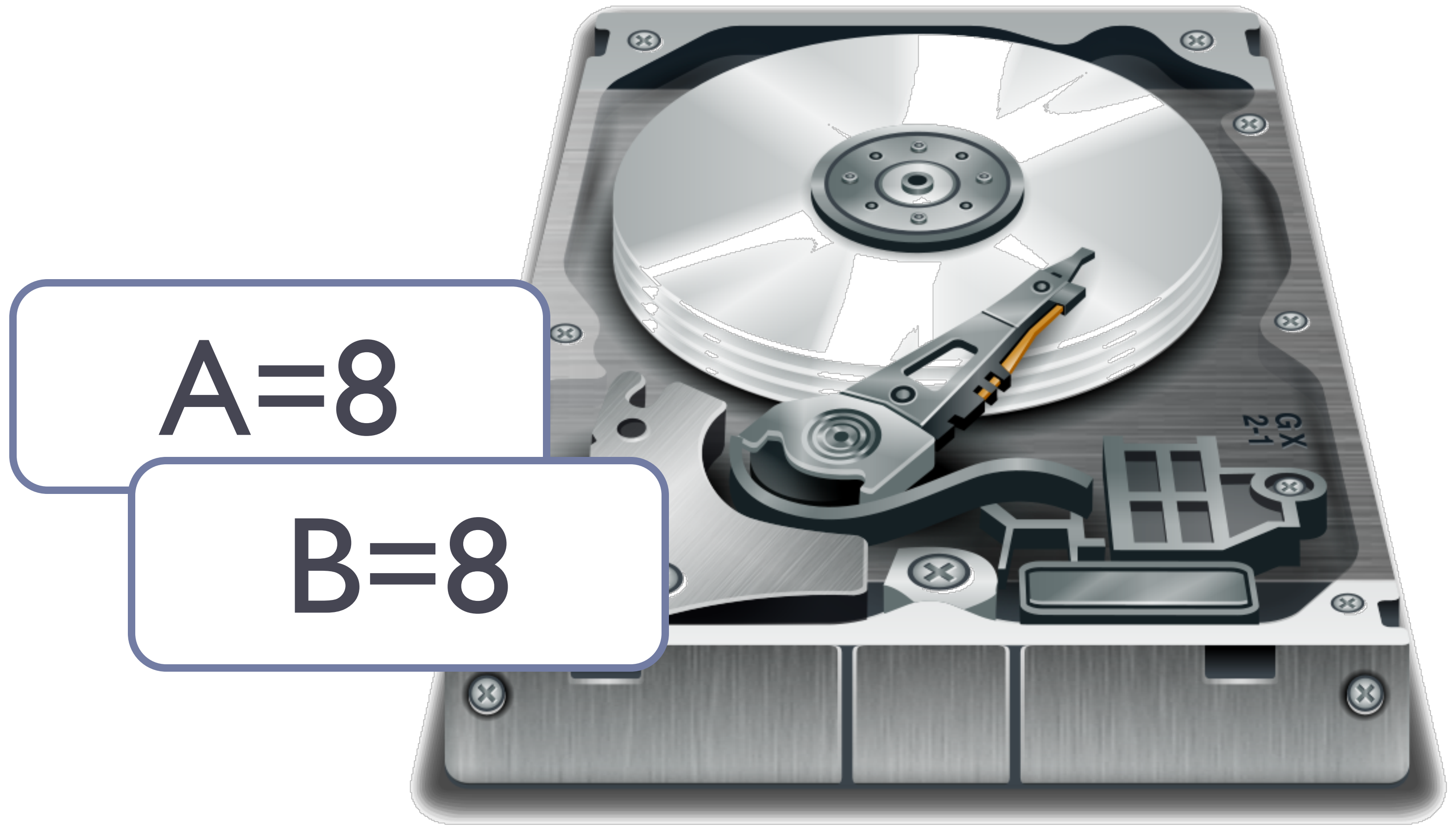
```
START TRANSACTION
READ(A,t);
t := t*2;
WRITE(A,t);
READ(B,t);
t := t*2;
WRITE(B,t);
COMMIT;
```

t=16

buffer

A=16

B=8



A=8

B=8

START TRANSACTION

READ(A,t);

t := t*2;

WRITE(A,t);

READ(B,t);

t := t*2;

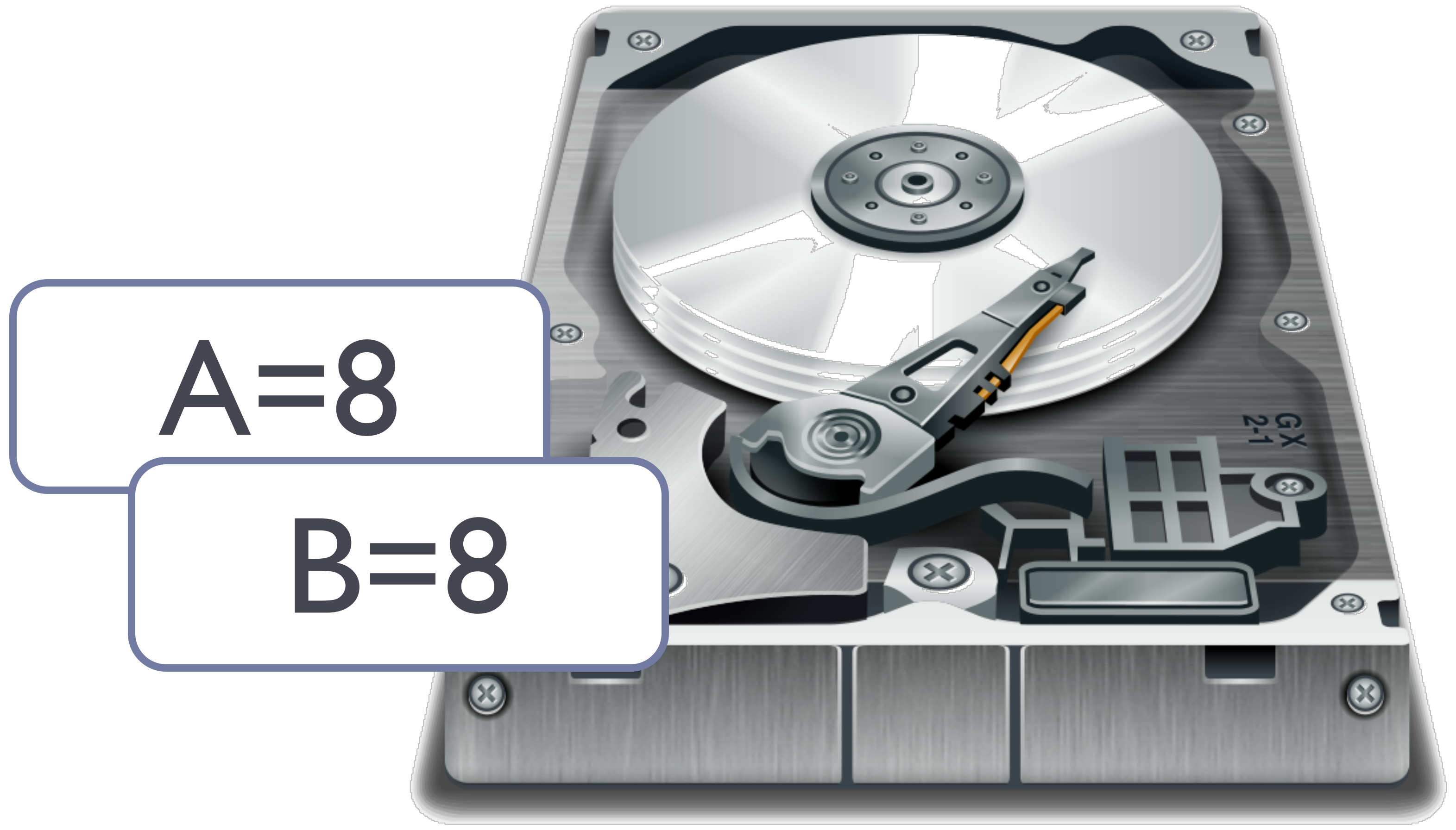
WRITE(B,t);

COMMIT;

t=8

buffer

A=16	B=16	



A=8

B=8

START TRANSACTION

READ(A,t);

t := t*2;

WRITE(A,t);

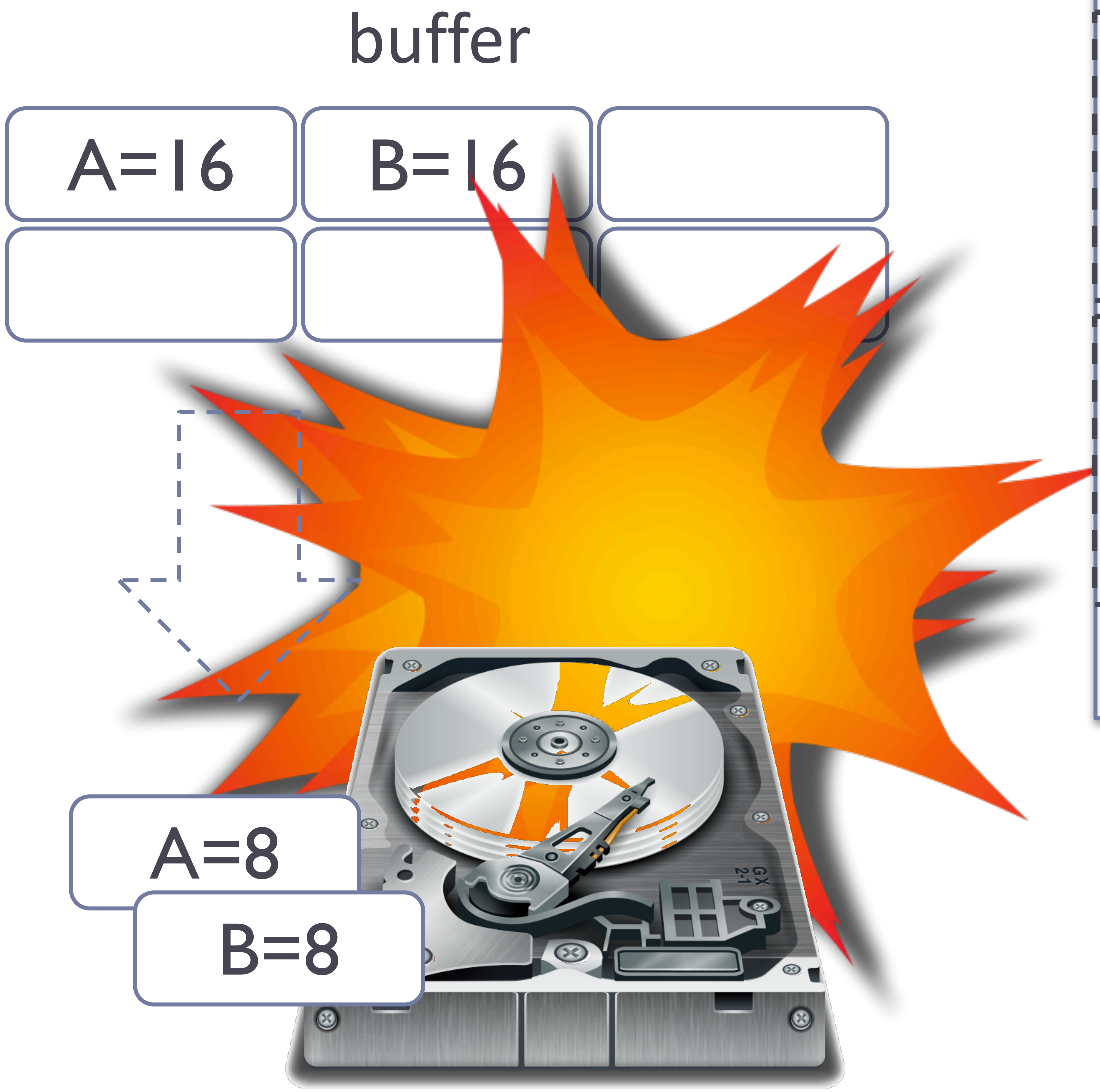
READ(B,t);

t := t*2;

WRITE(B,t);

COMMIT;

t=16



```
START TRANSACTION
READ(A,t);
t := t*2;
WRITE(A,t);
READ(B,t);
t := t*2;
WRITE(B,t);
COMMIT;
```

t=16

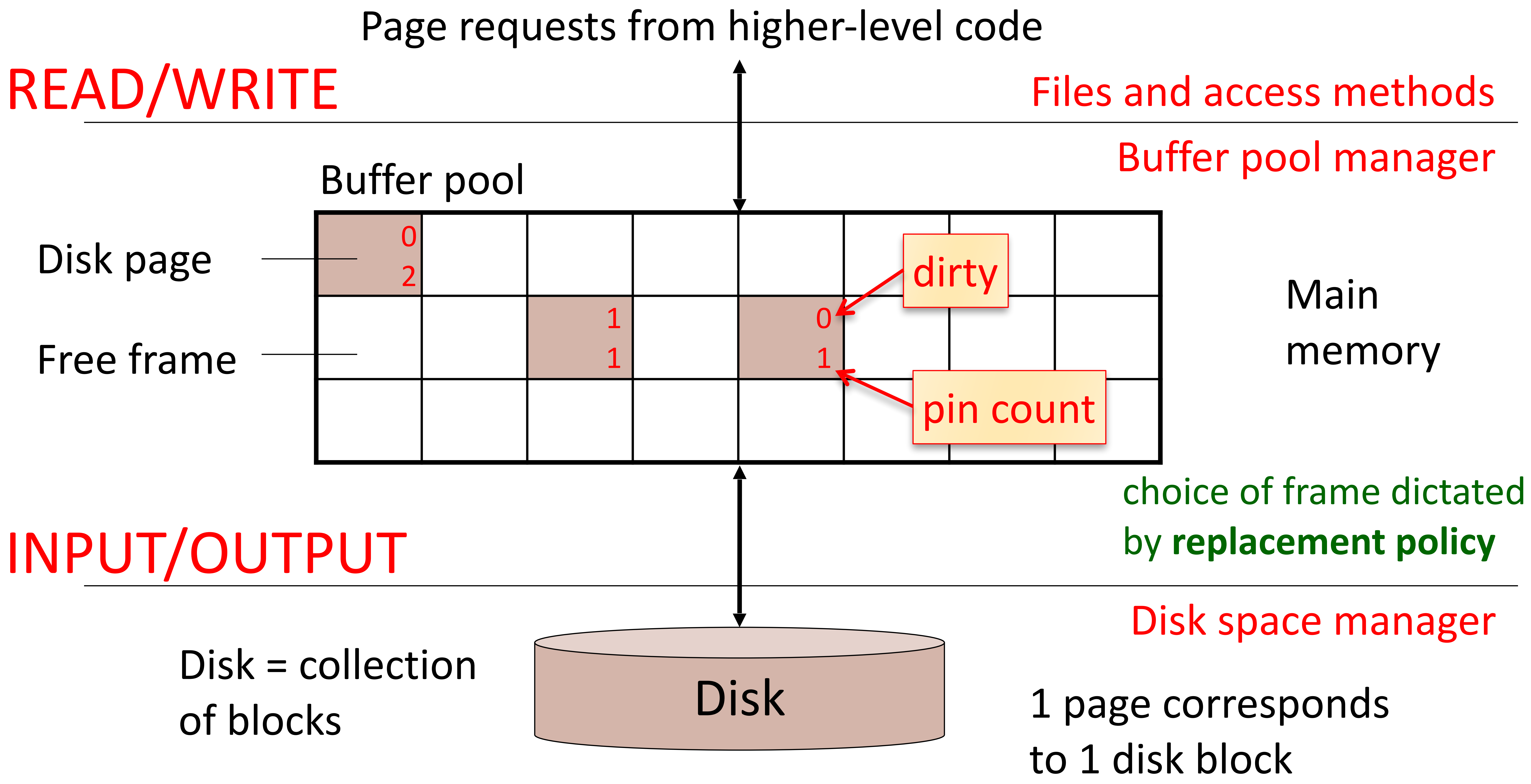
Solution: Use a Log

- Log = append-only file containing log records
- Note: multiple transactions run concurrently, log records are interleaved
- After a system crash, use log to:
  - Redo some transactions that did commit
  - Undo other transactions that did not commit

- Three kinds of logs: undo, redo, undo/redo

- WAL: Write Ahead Logging
  - All modification are written to a log before they are applied

Buffer Manager



Page requests from higher-level code

READ/WRITE

Files and access methods

Buffer pool manager

Buffer pool

Disk page

Free frame

dirty

pin count

Main memory

choice of frame dictated by **replacement policy**

INPUT/OUTPUT

Disk space manager

Disk = collection of blocks

Disk

1 page corresponds to 1 disk block

- Data must be in RAM for DBMS to operate on it!
- Buffer pool = table of <frame#, pageid> pairs

Buffer Manager Policies

- **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 or NO-FORCE**
  - Should all updates of a transaction be forced to disk before the transaction commits?

	No Steal	Steal
No Force		Fastest
Force	Slowest	

	No Steal	Steal
No Force	No UNDO REDO	UNDO REDO
Force	No UNDO No REDO	UNDO No REDO

ARIES Recovery Algorithm Overview

Three phases:

1. **Analysis**
   - Figure out what was going on at time of crash
   - List of dirty pages and active transactions
2. **Redo**
   - Redo all operations, even for transactions that will not commit
   - Get back to state at the moment of the crash
3. **Undo**
   - Remove effects of all uncommitted transactions
   - Log changes during undo in case of another crash during undo

Algorithms for Recovery and Isolation Exploiting Semantics

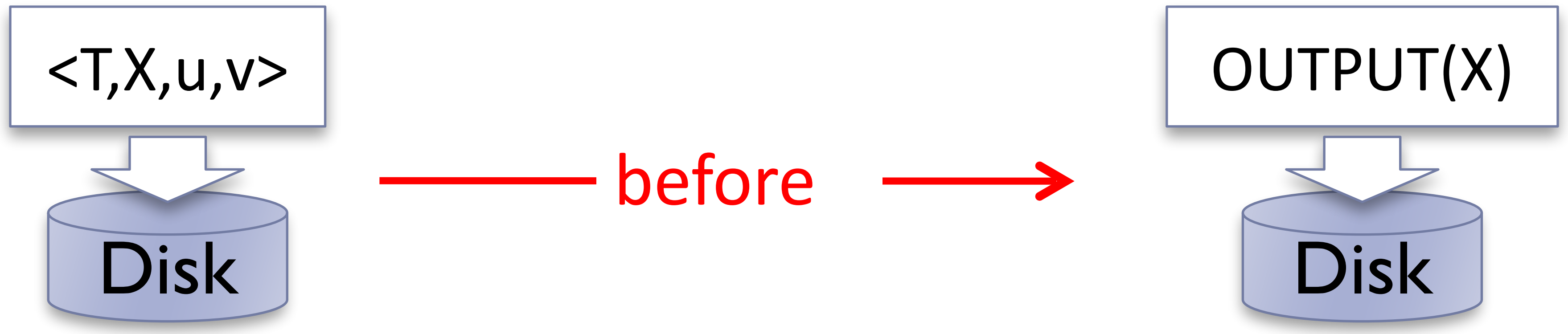
ARIES Recovery Algorithm Overview

Three principles:

1. **Write-Ahead Logging (WAL)**
    - Any change to a DB object is first recorded to the log
    - A log record must be written to disk before the corresponding object
2. **Repeating history**
    - Reinstate the exact state of the system before the crash
3. **Logging changes during UNDO**
    - Log UNDOs so we don’t repeat in a subsequent crash

Write-Ahead Log

1. Must force the log record of an update before the corresponding data page gets to disk



<T,X,u,v>

Disk

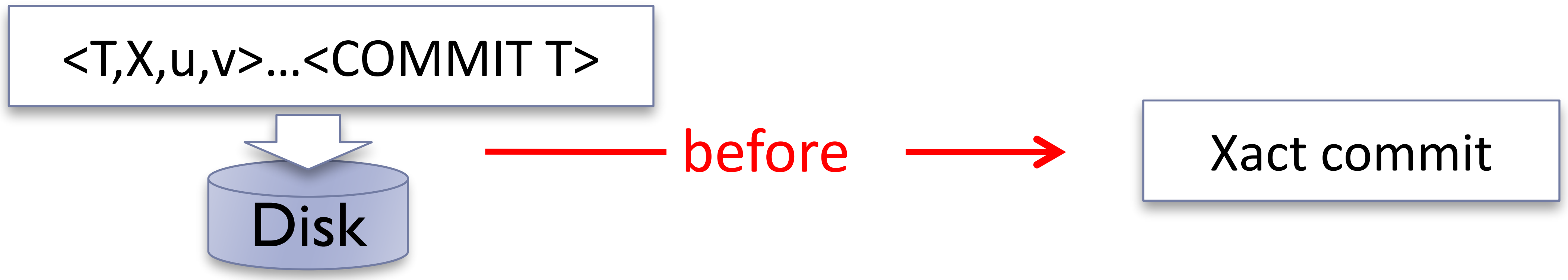
before

OUTPUT(X)

Disk

2. Must force all log records for a Xact before commit

- Xact is considered committed when its commit log record makes it to stable storage.



<T,X,u,v>…<COMMIT T>

Disk

before

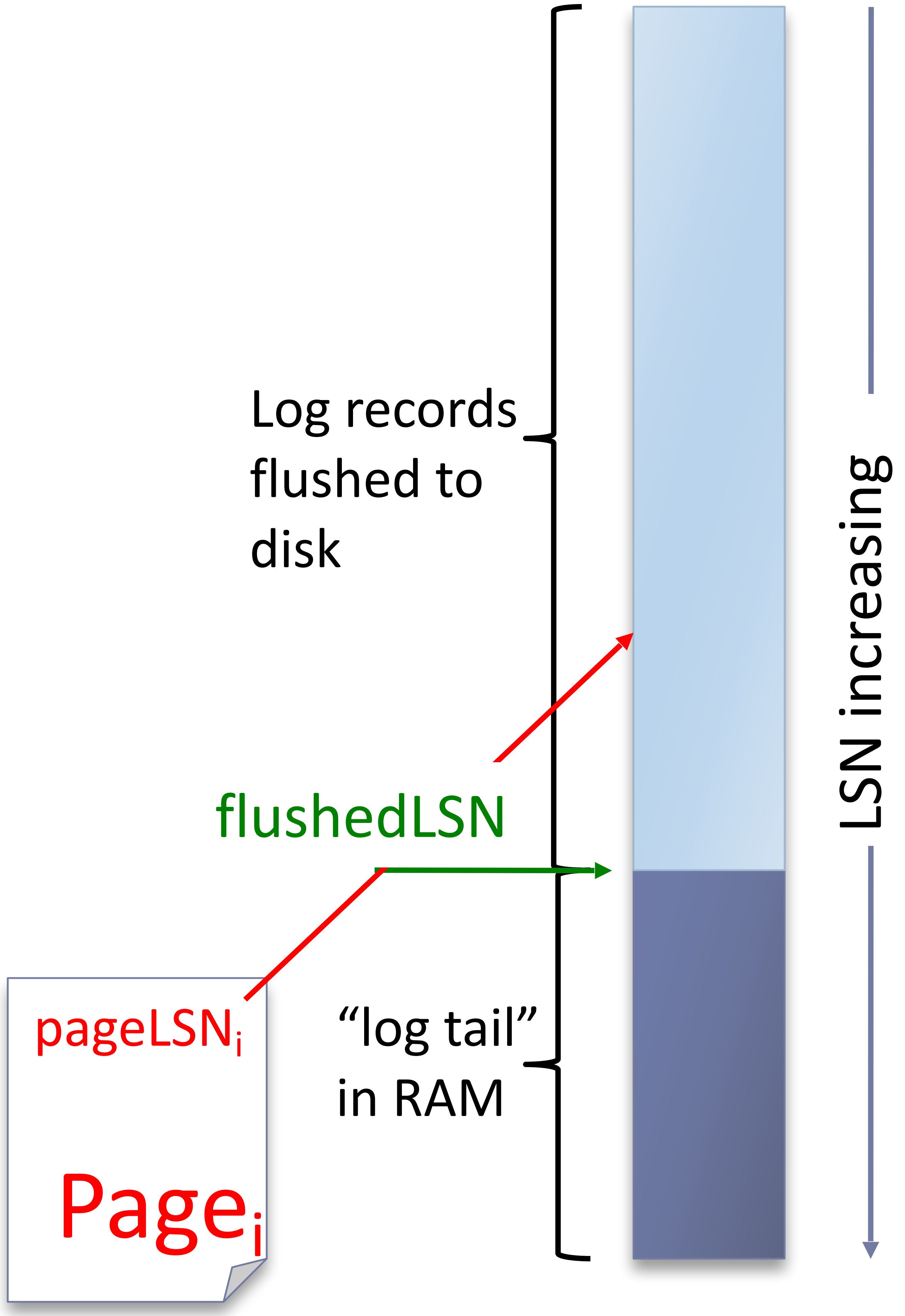
Xact commit

#1 (with UNDO info) helps guarantee atomicity

#2 (with REDO info) helps guarantee durability

The Log

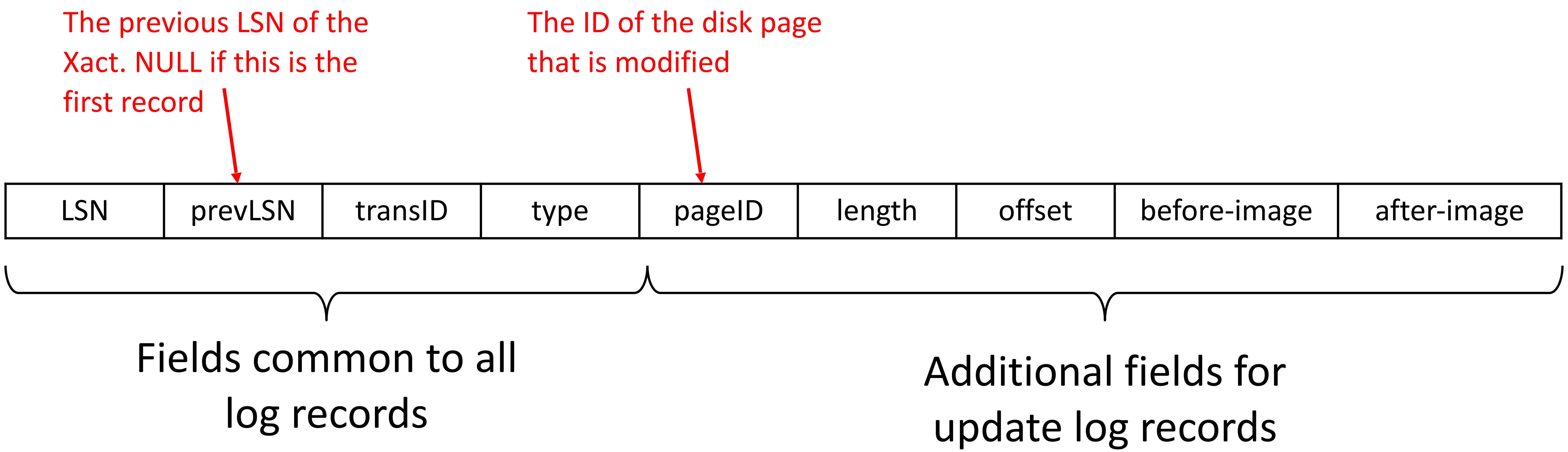
- Each log record has a unique
  Log Sequence Number (LSN)
  - Always increasing
- Each data page contains a pageLSN
  - The LSN of the most recent log record that updated that page
- System keeps track of flushedLSN
  - Max LSN flushed to stable storage



Types of Log Records

- Update
  - Whenever a page is modified, and update record is appended to the log tail
- Commit
  - When a Xact commits it force-writes a commit log record (i.e. flushes the log tail, up to and including this record). The Xact is considered committed the moment this record is on stable storage
- Abort
  - When a transaction is aborted (initiates rollback)
- End
  - When a Xact aborts or commits additional actions are initiated (e.g. rollback). Once those finish, an end record is appended
- CLR
  - Compensation Log Record: Logs the UNDOs
- Checkpoint

Log Records



The previous LSN of the Xact. NULL if this is the first record

The ID of the disk page that is modified

LSN	prevLSN	transID	type	pageID	length	offset	before-image	after-image

Fields common to all log records

Additional fields for update log records

- CLR records
  - REDO only：they do not get undone
    - Only contain after-image
  - Additional undoNextLSN field
    - Points to the next log record of the Xact that should be undone

Other Recovery-Related Structures

Transaction Table

transID	status	lastLSN

running/committing/aborting

The most recent log record for the Xact

Dirty Page Table

pageID	recLSN

First log entry that dirtied the page

Example of Recovery Structures

Transaction Table

transID	status	lastLSN

Dirty Page Table

pageID	recLSN

Buffer Pool

LSN	prevLSN	transID	type	pageID	length	offset	before-image	after-image
10	null	T1	update	P5	3	21	ABC	DEF

Example of Recovery Structures

Transaction Table

transID	status	lastLSN
T1	running	10

Dirty Page Table

pageID	recLSN
P5	10

Buffer Pool

P5 pageLSN=10	

LSN	prevLSN	transID	type	pageID	length	offset	before-image	after-image
10	null	T1	update	P5	3	21	ABC	DEF

Example of Recovery Structures

Transaction Table

transID	status	lastLSN
T1	running	10

Dirty Page Table

pageID	recLSN
P5	10

Buffer Pool

P5 pageLSN=10	

LSN	prevLSN	transID	type	pageID	length	offset	before-image	after-image
10	null	T1	update	P5	3	21	ABC	DEF
20	null	T2	update	P6	3	41	HIJ	KLM

Example of Recovery Structures

Transaction Table

transID	status	lastLSN
T1	running	10
T2	running	20

Dirty Page Table

pageID	recLSN
P5	10
P6	20

Buffer Pool

P5 pageLSN=10	P6 pageLSN=20

LSN	prevLSN	transID	type	pageID	length	offset	before-image	after-image
10	null	T1	update	P5	3	21	ABC	DEF
20	null	T2	update	P6	3	41	HIJ	KLM

Example of Recovery Structures

Transaction Table

transID	status	lastLSN
T1	running	10
T2	running	20

Dirty Page Table

pageID	recLSN
P5	10
P6	20

Buffer Pool

P5 pageLSN=10	P6 pageLSN=20

LSN	prevLSN	transID	type	pageID	length	offset	before-image	after-image
10	null	T1	update	P5	3	21	ABC	DEF
20	null	T2	update	P6	3	41	HIJ	KLM
30	20	T2	update	P5	3	20	GDE	QRS

Example of Recovery Structures

Transaction Table

transID	status	lastLSN
T1	running	10
T2	running	30

Dirty Page Table

pageID	recLSN
P5	10
P6	20

Buffer Pool

P5 pageLSN=30	P6 pageLSN=20

LSN	prevLSN	transID	type	pageID	length	offset	before-image	after-image
10	null	T1	update	P5	3	21	ABC	DEF
20	null	T2	update	P6	3	41	HIJ	KLM
30	20	T2	update	P5	3	20	GDE	QRS

Example of Recovery Structures

Transaction Table

transID	status	lastLSN
T1	running	10
T2	running	30

Dirty Page Table

pageID	recLSN
P5	10
P6	20

Buffer Pool

P5 pageLSN=30	P6 pageLSN=20

LSN	prevLSN	transID	type	pageID	length	offset	before-image	after-image
10	null	T1	update	P5	3	21	ABC	DEF
20	null	T2	update	P6	3	41	HIJ	KLM
30	20	T2	update	P5	3	20	GDE	QRS
40	10	T1	update	P7	3	21	TUV	WXY

Example of Recovery Structures

Transaction Table

transID	status	lastLSN
T1	running	40
T2	running	30

Dirty Page Table

pageID	recLSN
P5	10
P6	20
P7	40

Buffer Pool

P5 pageLSN=30	P6 pageLSN=20
	P7 pageLSN=40

LSN	prevLSN	transID	type	pageID	length	offset	before-image	after-image
10	null	T1	update	P5	3	21	ABC	DEF
20	null	T2	update	P6	3	41	HIJ	KLM
30	20	T2	update	P5	3	20	GDE	QRS
40	10	T1	update	P7	3	21	TUV	WXY

Normal Execution

- Update transaction table on Xact start/end
- For each update:
  - Create log record with LSN ℓ=++MaxLSN and prevLSN=TransTable[transID].lastLSN
  - Update TransTable[transID].lastLSN=ℓ
  - If modified page not in dirty table, add it with recLSN=ℓ
- If the buffer manager steals a dirty page, remove its entry from the DPT

Transaction Commit

- Write commit record to log
- Flush the log tail up to Xact's commit to disk
  - WAL rule #2: flushedLSN ≥ lastLSN
  - Note that log flushes are sequential, synchronous writes, so cheaper than forcing updated data
- Remove entry from the TransTable
- Write end record to log

Transaction Abort (no crash)

- Write abort log record before starting rollback
- "Play back" undoing all updates
  - Get lastLSN of Xact from the TransTable
  - Follow chain of log records via prevLSN
  - For each update encountered
    - Write a CLR for each undone operation with undoNextLSN = prevLSN of record being undone
    - Undo the operation (using the before-image of the log record)
- Remove entry from the TransTable
- Write end record to log

Checkpoints

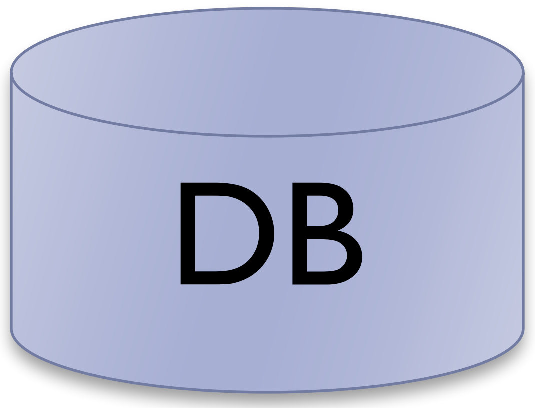
- begin_checkpoint
  - Indicates where checkpoint began
- end_checkpoint
  - Contains the Transaction Table and the Dirty Page Table as they were at begin_checkpoint
- Store the LSN of the most recent checkpoint at a master record on disk

The Big Picture: What's Where

log

Log Records

- LSN
- prevLSN
- transID
- type
- …

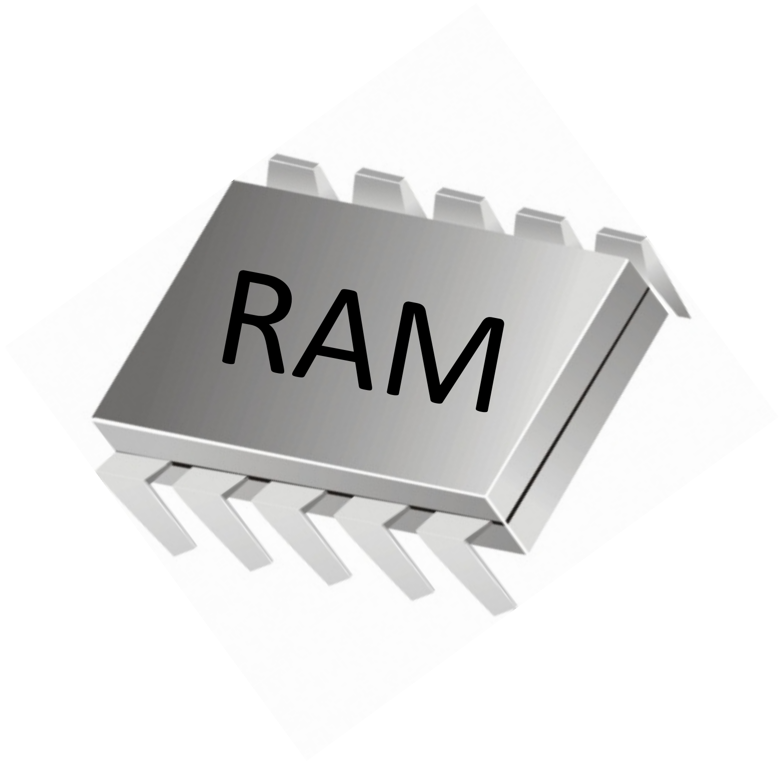


Data pages

Each with a
pageLSN

Master record

LSN of most
recent checkpoint



Transaction Table

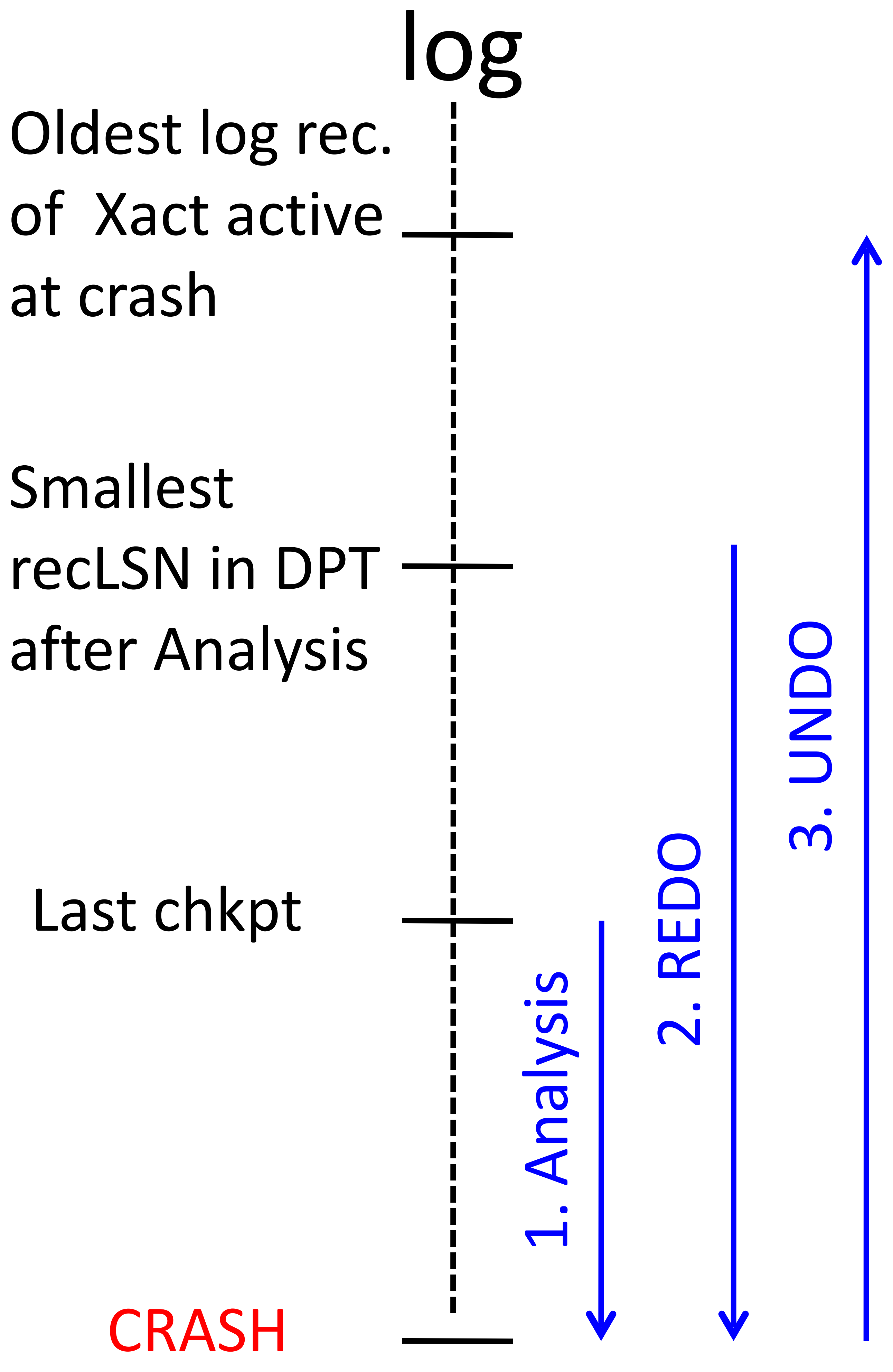
- lastLSN
- status

Dirty Page Table

- recLSN

flushedLSN

Crash Recovery: Big Picture



- Start from a checkpoint (found from master record)
- Three phases:
  1. Analysis – update structures
     - TransTable: active Xacts at crash
     - DBT: pages that *might* be dirty at crash
  2. REDO everything (repeat history)
     - Start at the smallest recLSN in DPT
  3. UNDO failed Xacts
     - Stop at the oldest LSN of active Xact

Phase 1: Analysis

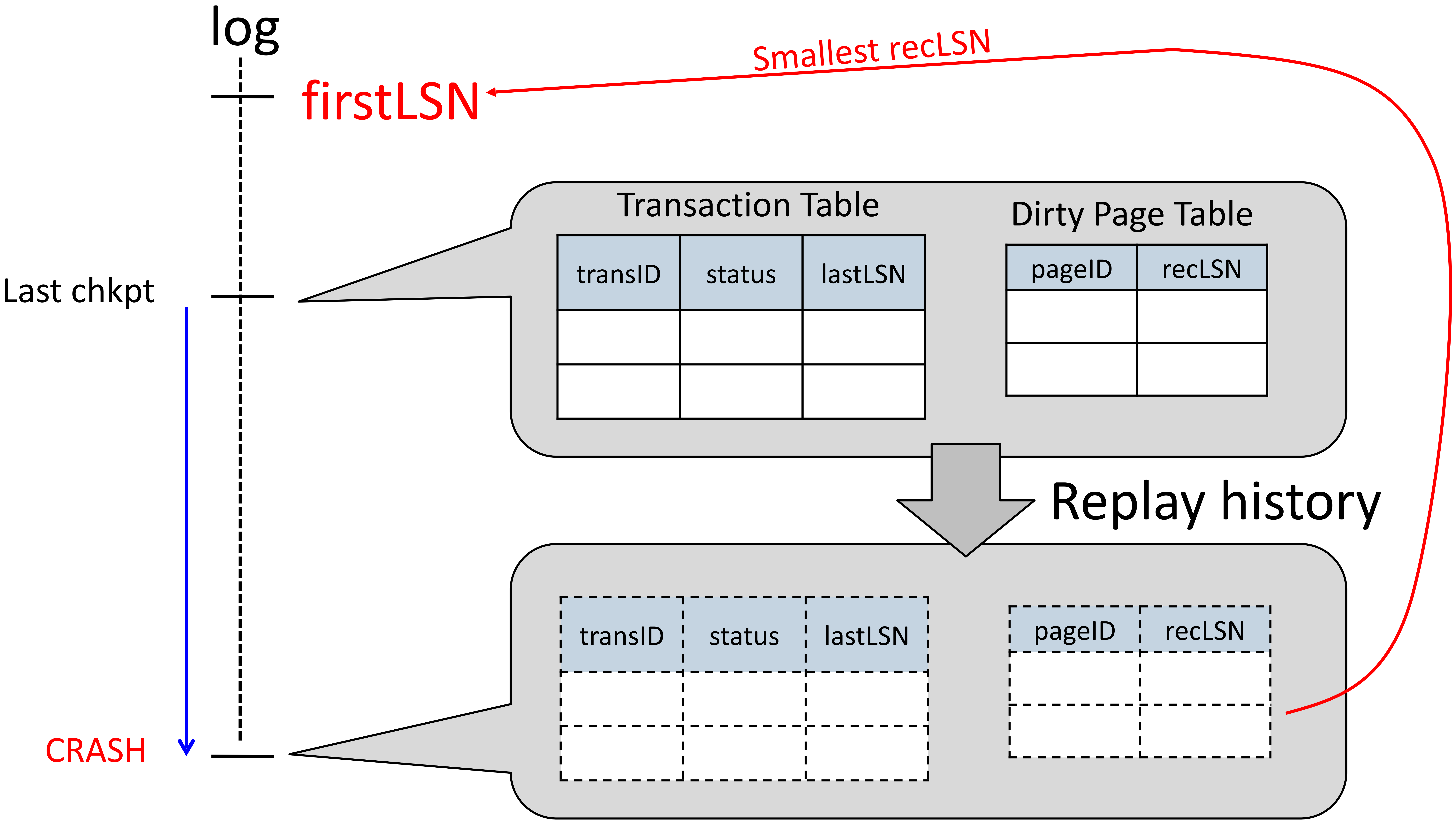
- Goal
  - Determine point in log where to start REDO
  - Determine set of dirty pages when crashed
    - Conservative estimate
  - Identify active transactions when crashed (loser transactions)
- Approach
  - Rebuild active transactions table and dirty pages table
  - Compute: firstLSN = smallest of all recLSN in DPT

Phase 1: Analysis

- Load the Transaction Table and Dirty Page Table stored at the checkpoint

- Scan log forward from checkpoint
  - end record: remove Xact from TransTable
  - All other records:
    - add Xact to TransTable (if not there)
    - Set lastLSN=LSN
    - Change status accordingly
  - update record: if P not in DPT, add it with recLSN=LSN

Phase 1: Analysis



log

firstLSN

Smallest recLSN

Last chkpt

CRASH

Transaction Table

transID	status	lastLSN

Dirty Page Table

pageID	recLSN

Replay history

transID	status	lastLSN

pageID	recLSN

Phase 2: REDO

Principles:

- Scan the log forward from firstLSN ← Why start here?
- Read all records sequentially, and reapply all updates
- Do not record REDO actions in the log
- Needs the DPT

Phase 2: REDO

Details:

- For each updateable record (update or CLR) REDO the action, unless:
  - Affected page not in DPT
  - Affected page in DPT but recLSN > LSN
  - pageLSN (in DB) ≥ LSN (requires I/O)

- To REDO:
  - Reapply logged action
  - Set pageLSN to LSN

Phase 3: UNDO

Principles:

- Start from the end of the log, move backwards
- Read only affected log entries (loser Xacts)
- Undo actions logged as special entries: CLR (Compensation Log Records)
- CLRs are redone, but never undone

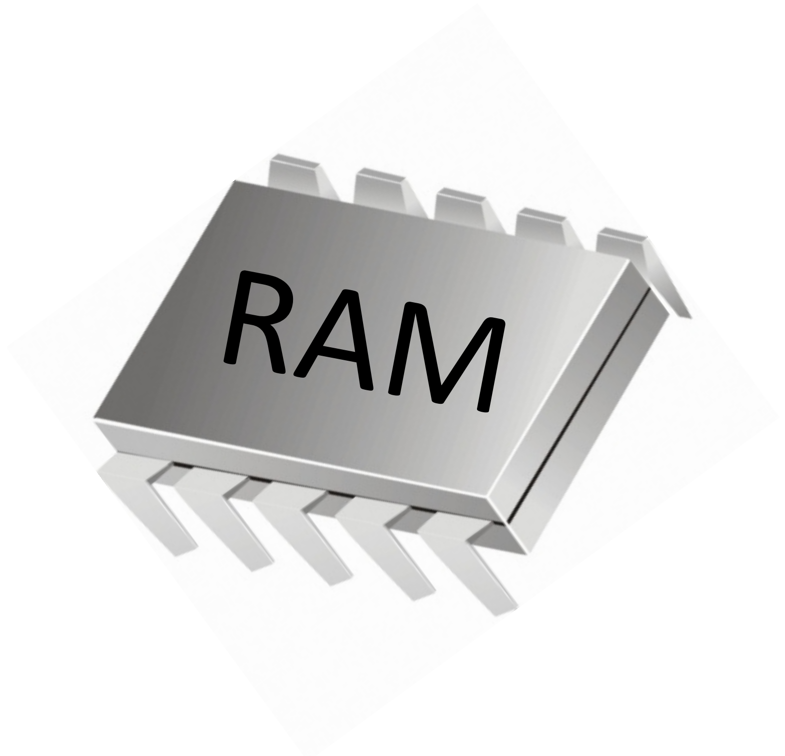
Phase3 : UNDO

Details:

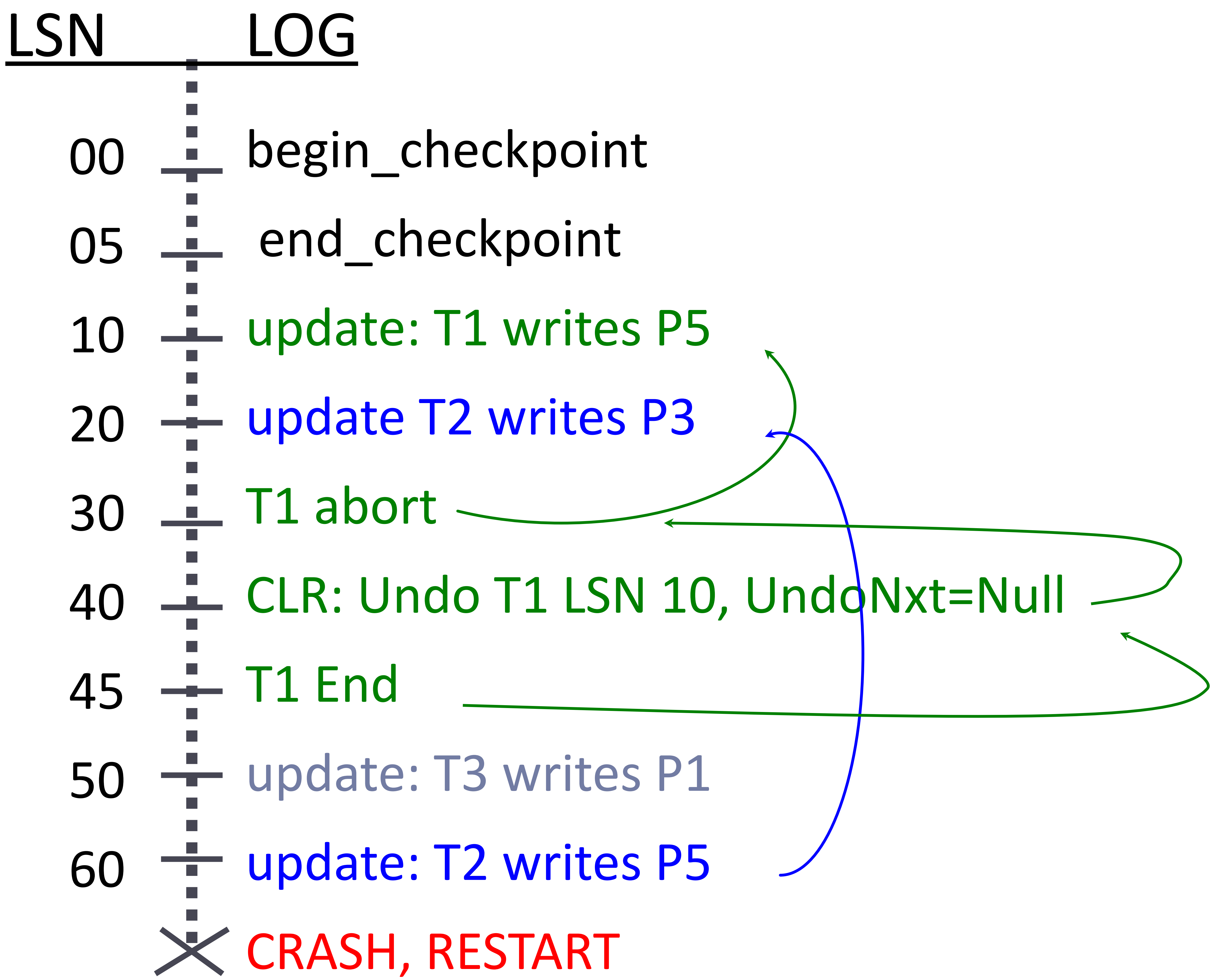
- Loser Xacts : all Xacts in the Transaction Table
- ToUndo = {lastLSN of all Loser Xacts}

- While ToUndo is not empty :
  - Choose the most recent (largest) LSN in ToUndo
  - If LSN is a CLR and undoNextLSN=null
    - Write end record for Xact
  - If LSN is a CLR and undoNextLSN ≠ null
    - Add undoNextLSN to ToUndo
  - If LSN is an update
    - Undo the action
    - Write a CLR
    - Add prevLSN to ToUndo

Example of Recovery – (up to crash)



- Xact Table
  - lastLSN
  - status
- Dirty Page Table
  - recLSN
- flushedLSN



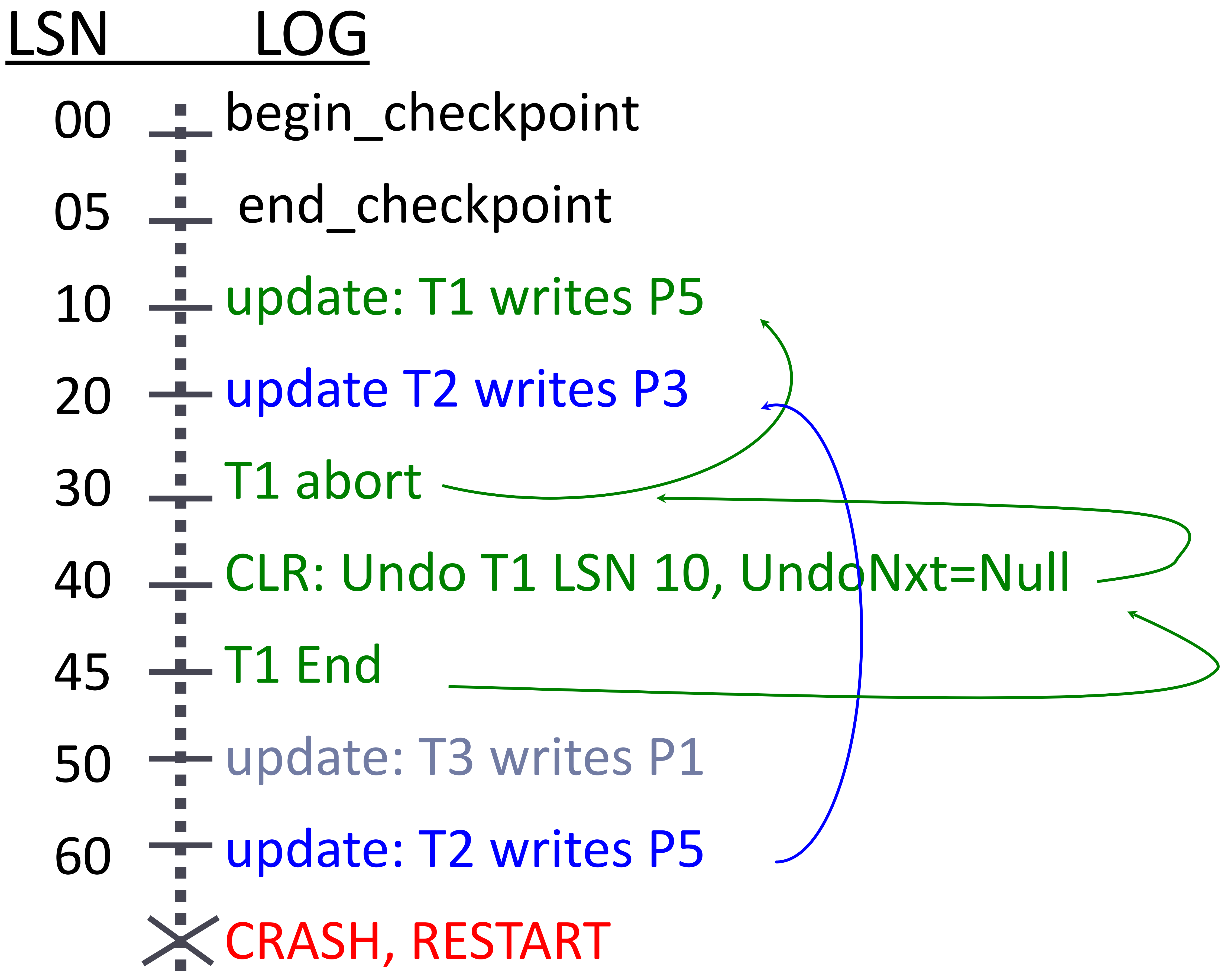
LSN	LOG
00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40	CLR: Undo T1 LSN 10, UndoNxt=Null
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART

Trans Table

Trans	lastLSN	Stat
T2	60	r
T3	50	r

Dirty Page Table

PageId	recLSN
P5	10
P3	20
P1	50



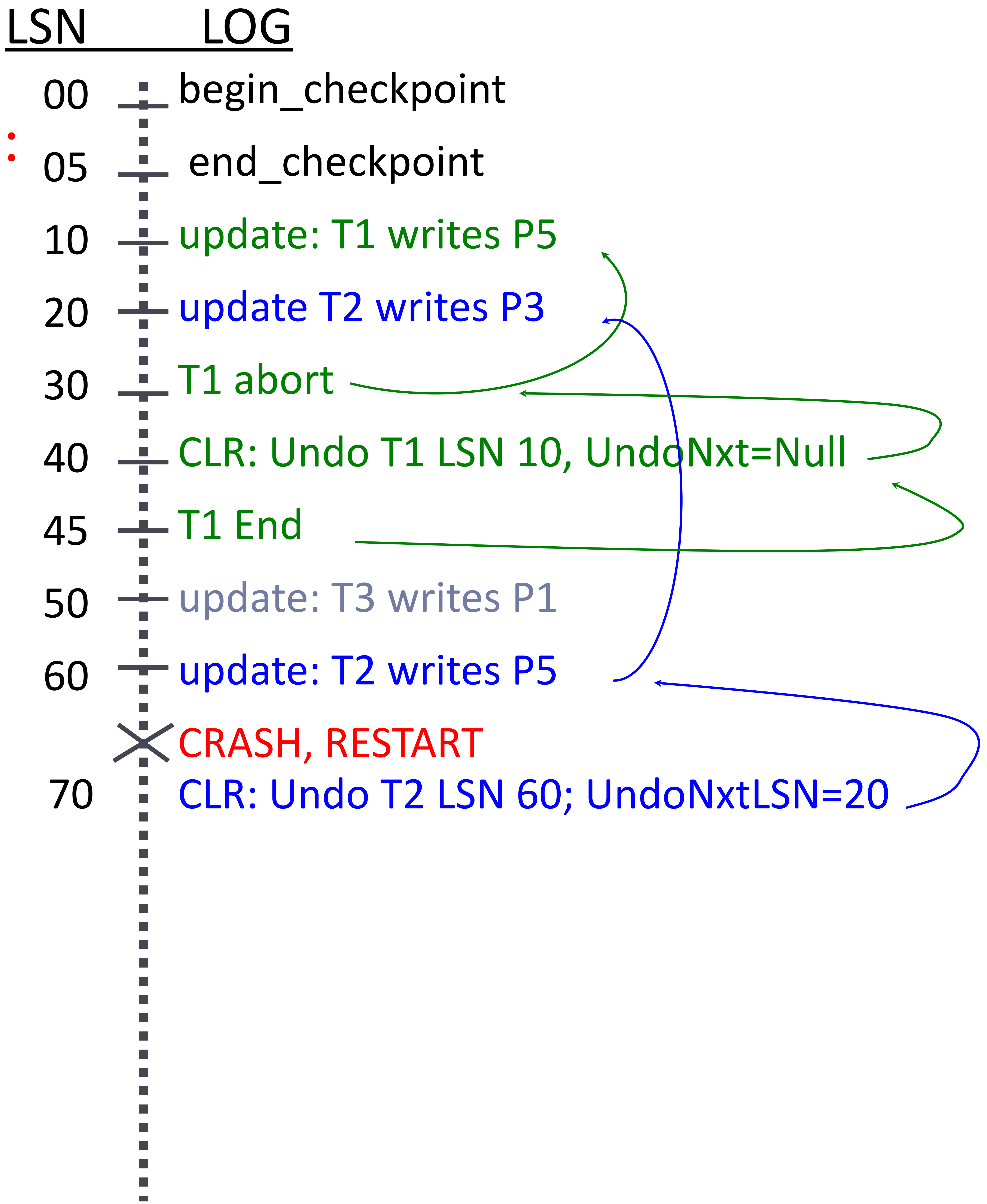
LSN	LOG
00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40	CLR: Undo T1 LSN 10, UndoNxt=Null
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART

Redo starts at LSN 10;
in this case, reads P5, P3, and P1 from disk, redoes ops if pageLSN < LSN

ToUndo set initializes to {50,60}

After Analysis & REDO:

ToUndo: {50, 60}

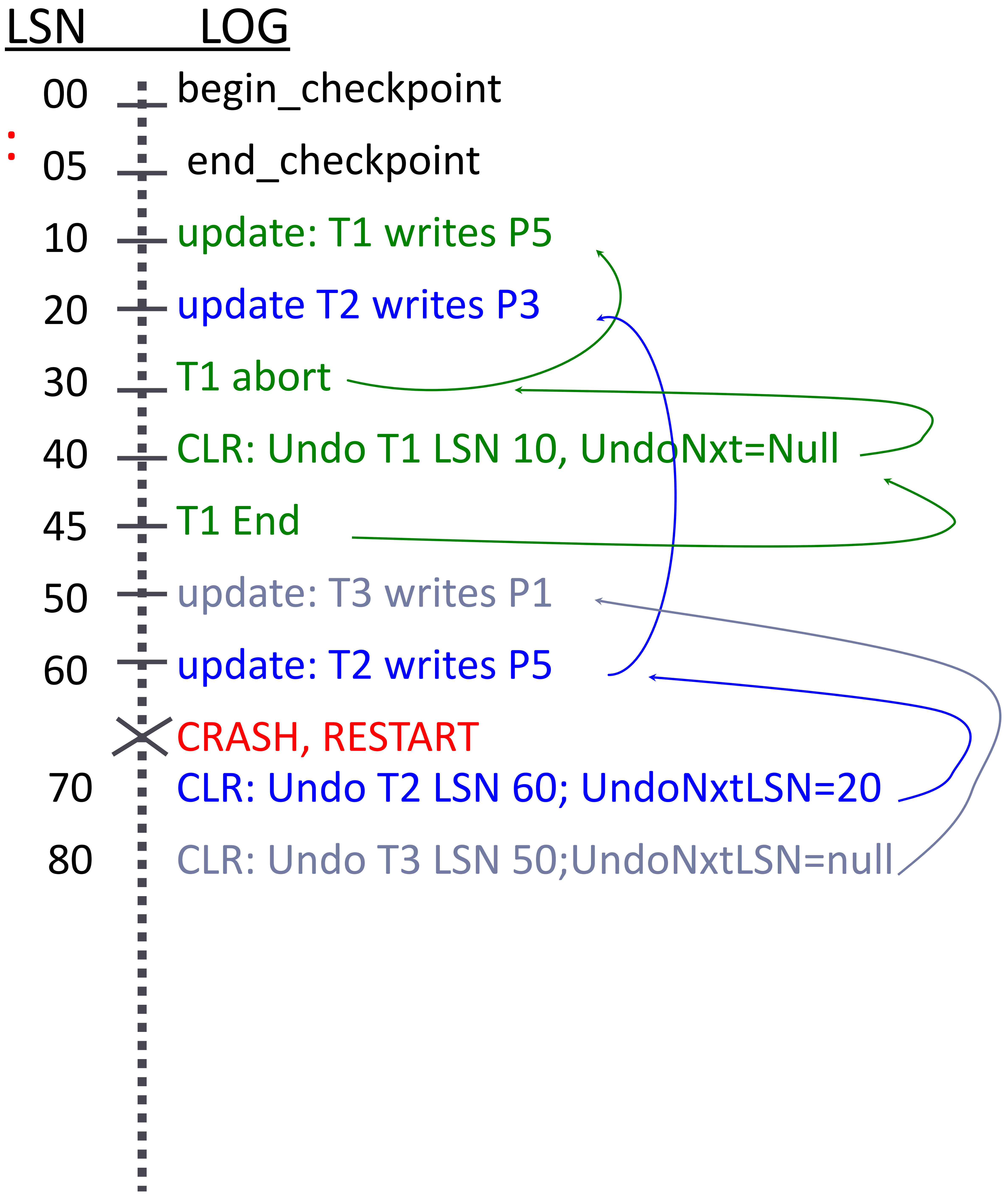


LSN	LOG
00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40	CLR: Undo T1 LSN 10, UndoNxt=Null
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART
70	CLR: Undo T2 LSN 60; UndoNxtLSN=20

After Analysis & REDO:

ToUndo: {50, 60}

ToUndo: {50, 20}



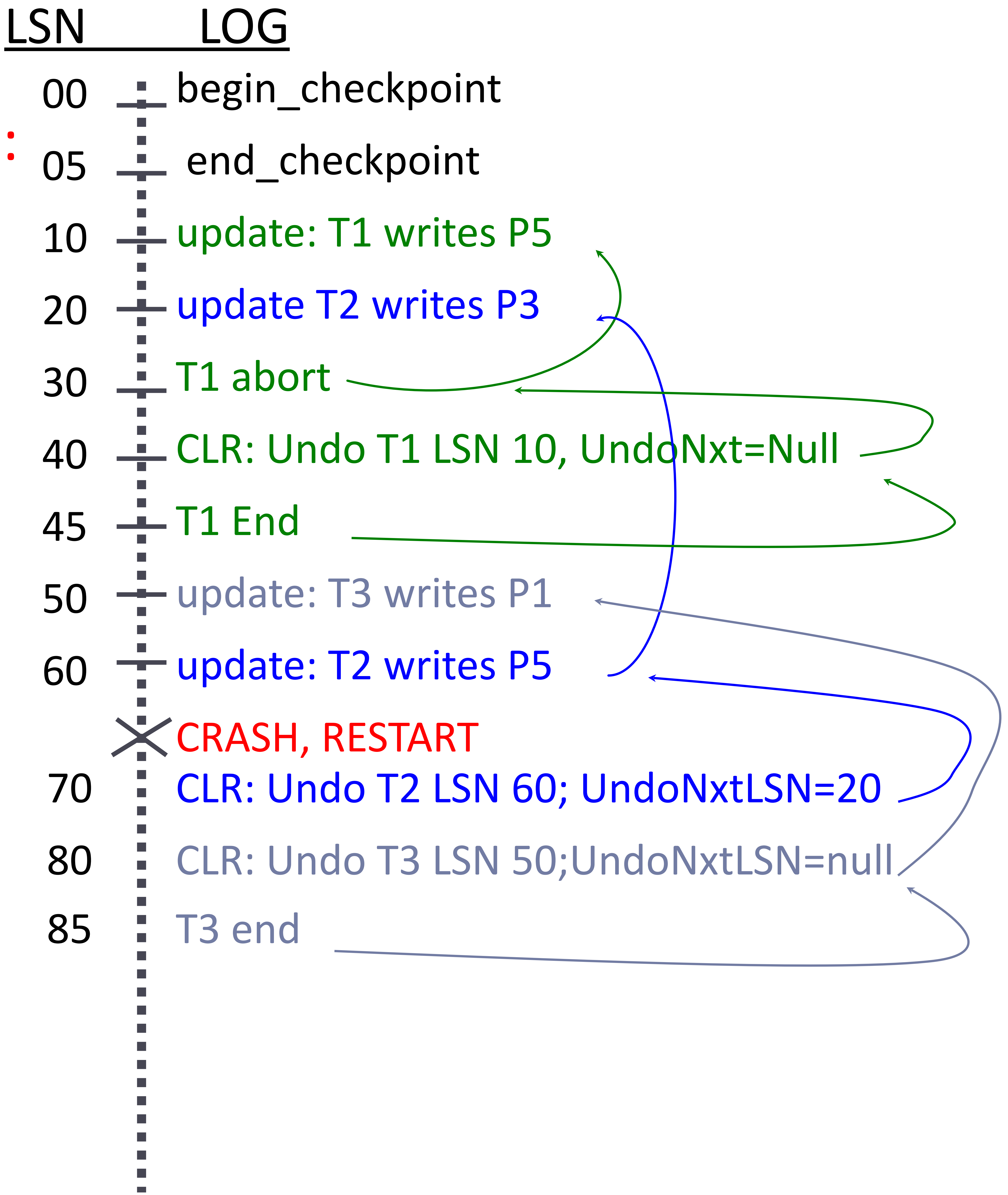
LSN	LOG
00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40	CLR: Undo T1 LSN 10, UndoNxt=Null
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART
70	CLR: Undo T2 LSN 60; UndoNxtLSN=20
80	CLR: Undo T3 LSN 50;UndoNxtLSN=null

After Analysis & REDO:

ToUndo: {50, 60}

ToUndo: {50, 20}

ToUndo: {20}



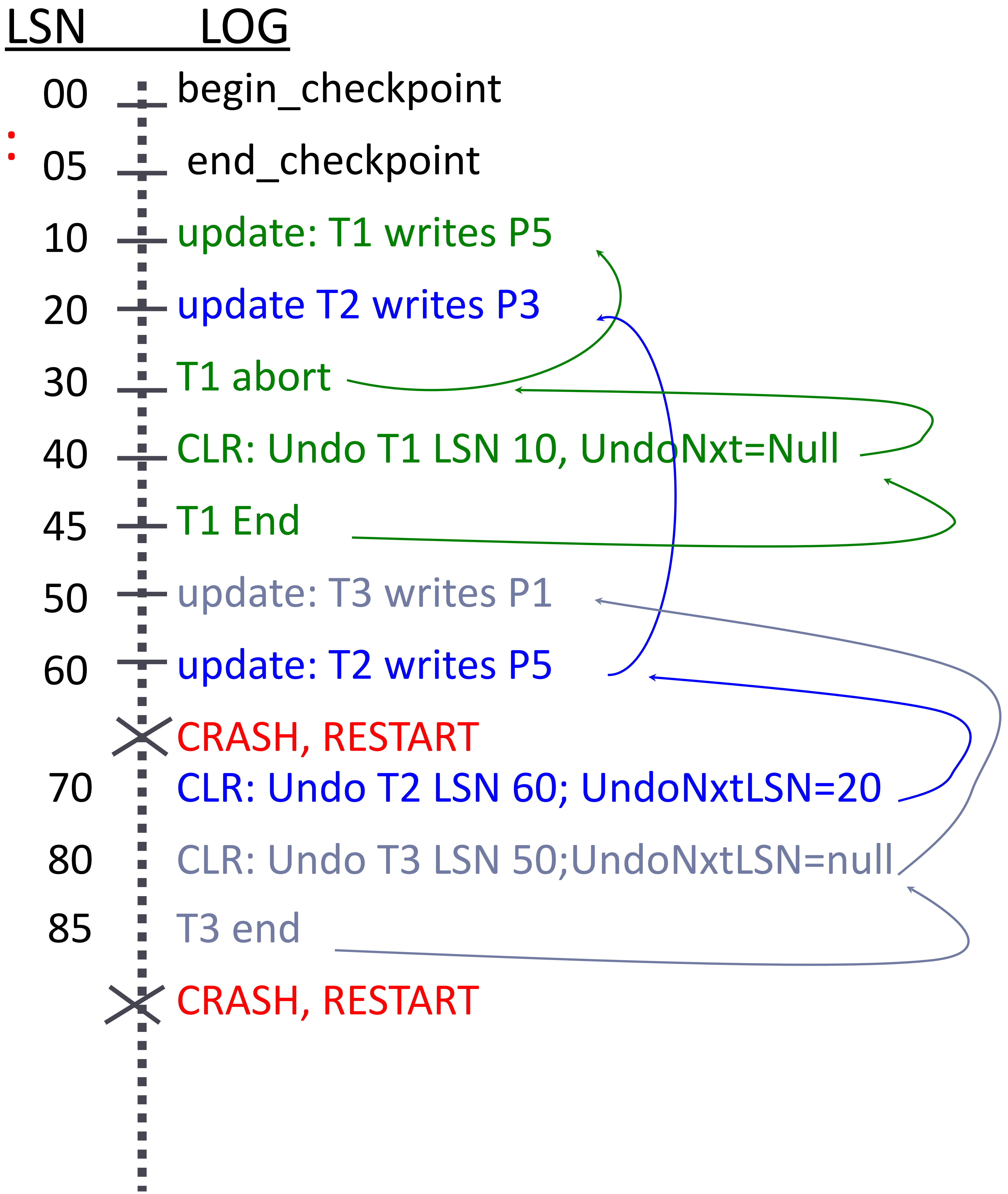
LSN	LOG
00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40	CLR: Undo T1 LSN 10, UndoNxt=Null
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART
70	CLR: Undo T2 LSN 60; UndoNxtLSN=20
80	CLR: Undo T3 LSN 50;UndoNxtLSN=null
85	T3 end

After Analysis & REDO:

ToUndo: {50, 60}

ToUndo: {50, 20}

ToUndo: {20}



LSN	LOG
00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40	CLR: Undo T1 LSN 10, UndoNxt=Null
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART
70	CLR: Undo T2 LSN 60; UndoNxtLSN=20
80	CLR: Undo T3 LSN 50;UndoNxtLSN=null
85	T3 end
	CRASH, RESTART

After Analysis & REDO:

ToUndo: {50, 60}

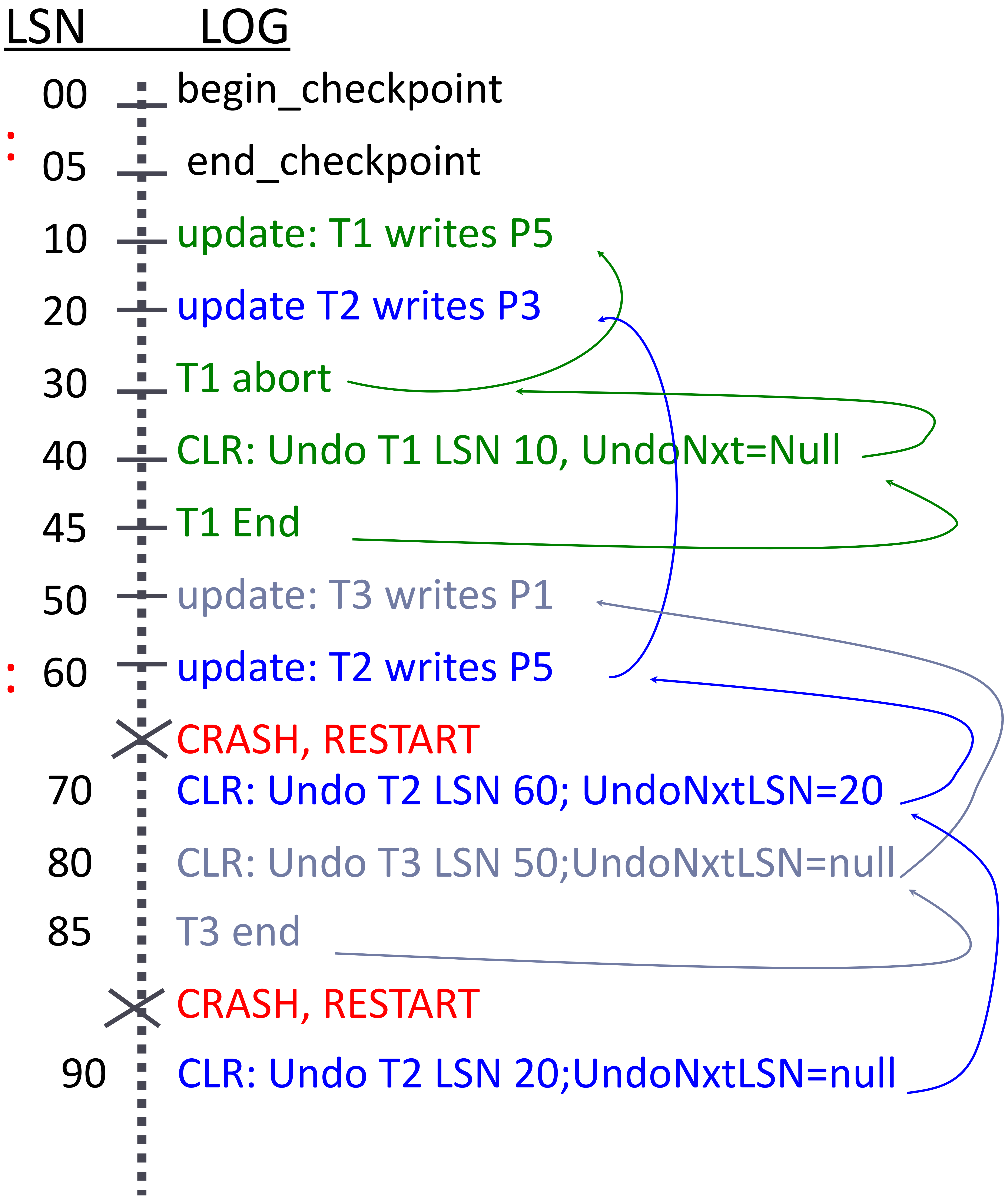
ToUndo: {50, 20}

ToUndo: {20}

After Analysis & REDO:

ToUndo: {70}

ToUndo: {20}



LSN	LOG
00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40	CLR: Undo T1 LSN 10, UndoNxt=Null
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART
70	CLR: Undo T2 LSN 60; UndoNxtLSN=20
80	CLR: Undo T3 LSN 50;UndoNxtLSN=null
85	T3 end
	CRASH, RESTART
90	CLR: Undo T2 LSN 20;UndoNxtLSN=null

After Analysis & REDO:

ToUndo: {50, 60}

ToUndo: {50, 20}

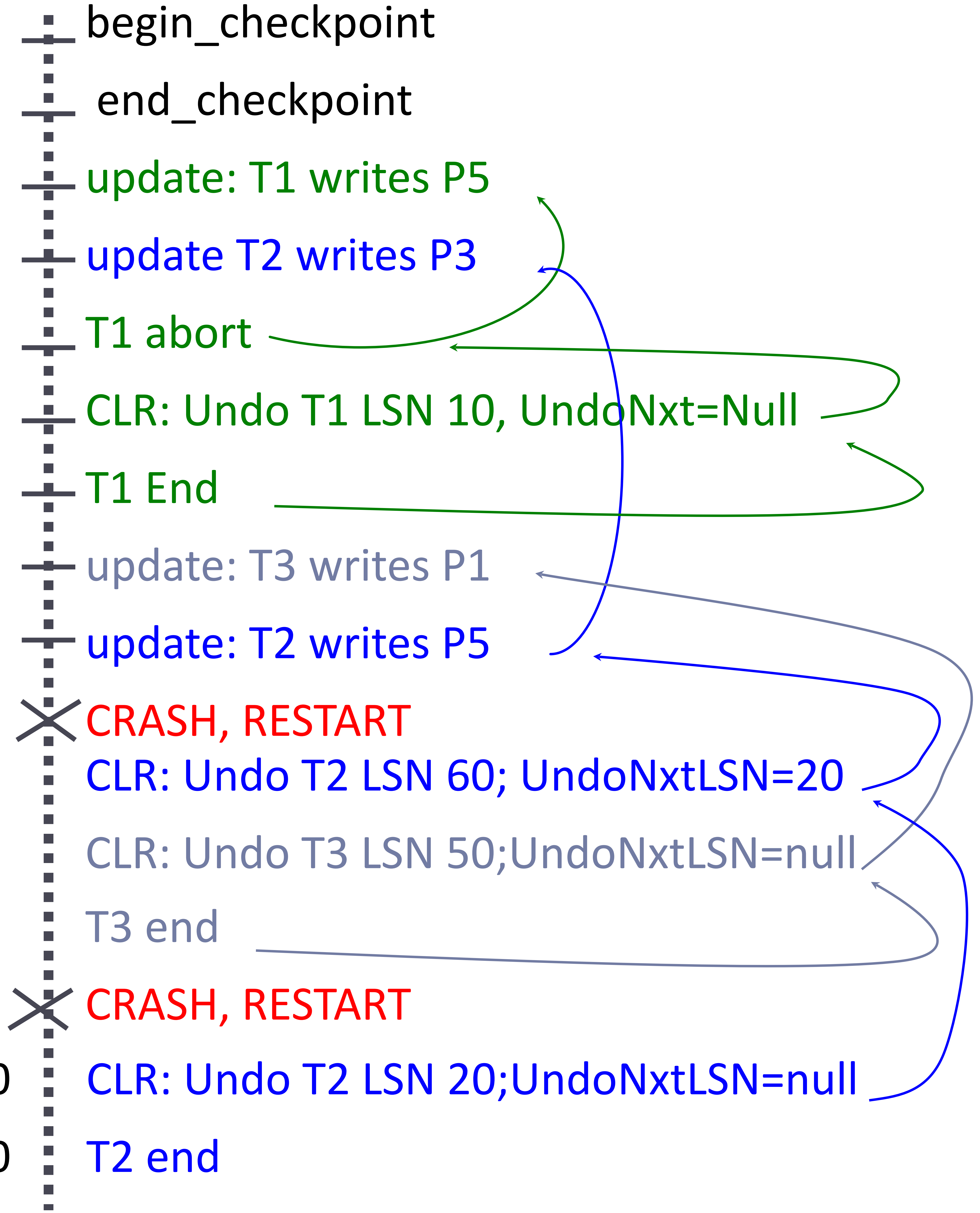
ToUndo: {20}

After Analysis & REDO:

ToUndo: {70}

ToUndo: {20}

ToUndo: {}



LSN	LOG
00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40	CLR: Undo T1 LSN 10, UndoNxt=Null
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART
70	CLR: Undo T2 LSN 60; UndoNxtLSN=20
80	CLR: Undo T3 LSN 50;UndoNxtLSN=null
85	T3 end
	CRASH, RESTART
90	CLR: Undo T2 LSN 20;UndoNxtLSN=null
100	T2 end

Discussion

- What if we crash during Analysis? During REDO?
- How can we reduce the amount of work in Analysis?
- How do we reduce the amount of work in REDO?
- What affects the amount of work in UNDO?