

Database Management Systems

Lecture 4

Crash Recovery

Recovery - ACID

- the Recovery Manager (RM) in a DBMS ensures two important properties of transactions:
 - atomicity - the effects of uncommitted transactions (i.e., transactions that do not commit prior to the crash) are undone
 - durability - the effects of committed transactions survive system crashes



* T1, T2 commit before the crash, whereas T3 and T4 are still active when the system crashes

- when the system comes back up:
 - the effects of T1 & T2 must persist
 - T3 & T4 are undone (their effects are not persisted in the DB)

Transaction Failure - Causes

- system failure (hardware failures, bugs in the operating system, database system, etc)
 - all running transactions terminate
 - contents of internal memory – affected (i.e., lost)
 - contents of external memory – not affected
- application error (“bug”, e.g., division by 0, infinite loop, etc)
 - => transaction fails; it should be executed again only after the error is corrected
- action by the Transaction Manager (TM)
 - e.g., deadlock resolution scheme
 - a transaction is chosen as the deadlock victim and terminated
 - the transaction might complete successfully if executed again

Transaction Failure - Causes

- self-abort
 - based on some computations, a transaction can decide to terminate and undo its actions
 - there are special statements for this purpose, e.g., *ABORT*, *ROLLBACK*
 - it can be seen as a special case of *action by the TM*

Normal Execution

- during normal execution, transactions read / write database objects
- reading database object O:
 - bring O from the disk into a frame in the Buffer Pool (BP)*
 - copy O's value into a program variable
- writing database object O:
 - modify an in-memory copy of O (in the BP)
 - write the in-memory copy to disk

* see the *Databases* course in the 1st semester (lecture 8 - Buffer Manager)

Writing Objects

- options for the Buffer Manager (BM): *steal / no-steal, force / no-force*
- transaction T changes object O (in frame F in the BP)
- transaction T2 needs a page; the BM chooses F as a replacement frame (while T is still in progress)
 - *steal* approach:
 - T's changes can be written to disk while T is in progress (T2 steals a frame from T)
 - *no-steal* approach:
 - T's changes cannot be written to disk before T commits
- *force* approach
 - T's changes are immediately forced to disk when T commits
- *no-force* approach
 - T's changes are not forced to disk when T commits

Writing Objects

- *no-steal* approach
 - advantage - changes of aborted transactions don't have to be undone (such changes are never written to disk!)
 - drawback - assumption: all pages modified by active transactions can fit in the BP
- *force* approach
 - advantage - actions of committed transactions don't have to be redone
 - by contrast, when using *no-force*, the following scenario is possible: transaction T commits at time t_0 ; its changes are not immediately forced to disk; the system crashes at time $t_1 \Rightarrow$ T's changes have to be redone!
 - drawback - can result in excessive I/O
- *steal, no-force* approach – used by most systems

ARIES

- recovery algorithm; *steal, no-force* approach
- system restart after a crash - three phases:
 - analysis – determine:
 - active transactions at the time of the crash
 - *dirty pages*, i.e., pages in BP whose changes have not been written to disk
 - redo - reapply all changes (starting from a certain record in the log), i.e., bring the DB to the state it was in when the crash occurred
 - undo - undo changes of uncommitted transactions
- fundamental principle - *Write-Ahead Logging*
 - a change to an object O is first recorded in the log (e.g., in log record LR)
 - LR must be written to disk before the change to O is written to disk

ARIES

* example

- analysis
 - active transactions at crash time: T1, T3 (to be undone)
 - committed transactions: T2 (its effects must persist)
 - potentially dirty pages: P1, P2, P3
- redo
 - reapply all changes in order (1, 2, ...)
- undo
 - undo changes of T1 and T3 in reverse order (6, 5, 1)

LSN	Log
1	T1 writes P1
2	T2 writes P2
3	T2 commit
4	T2 end
5	T3 writes P3
6	T3 writes P2
crash & restart	

The Log (journal)

- history of actions executed by the DBMS
- stored in *stable storage*, i.e., keep ≥ 2 copies of the log on different disks (locations) - ensures the “durability” of the log
- records are added to the end of the log
- *log tail* - the most recent fragment of the log
 - kept in main memory and periodically forced to stable storage
- *Log Sequence Number* (LSN) - unique id for every log record
 - monotonically increasing (e.g., address of 1st byte of log record)
- every page P in the DB contains the *pageLSN*: the LSN of the most recent record in the log describing a change to P
- *log record* – fields:
 - prevLSN – linking a transaction’s log records
 - transID – id of the corresponding transaction
 - type – type of the log record

The Log

- each of the following actions results in a log record being written: *update page, commit, abort, end, undoing an update*
- update page P
 - add an *update type* log record ULR to the log tail (with LSN_{ULR})
 - $pageLSN(P) := LSN_{ULR}$
- transaction T commits
 - add a *commit type* log record CoLR to the log tail
 - force log tail to stable storage (including CoLR)
 - complete subsequent actions (remove T from transaction table)
- transaction T aborts
 - add an *abort type* log record to the log
 - initiate Undo for T
- transaction T ends
 - T commits / aborts - complete required actions

The Log

- transaction T ends
 - add an *end type* log record to the log
- undoing an update
 - i.e., when the change described in an update log record is undone
 - add a *compensation log record* (CLR) to the log
- obs. committed transaction – a transaction whose log records (including the *commit log record*) have been written to stable storage
- *update log record* – additional fields: *pageID* (id of the changed page), *length* (length of the change (in bytes)), *offset* (offset of the change), *before-image* (value before the change), *after-image* (value after the change)
 - can be used to undo / redo the change


The Log

- *compensation log record*
 - let U be an update log record describing an update of transaction T
 - let C be the compensation log record for U, i.e., C describes the action taken to undo the changes described by U
 - C has a field named *undoNextLSN*:
 - the LSN of the next log record to be undone for T
 - set to the value of prevLSN in U

* example: undo T10's update to P10

=> CLR with *transID* = T10,
pageID = P10, *length* = 2,
offset = 10,
before-image = JH and

undoNextLSN = LSN of 1st log record (i.e., the next record that is to be undone for transaction T10)



prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB

log

The Transaction Table and the Dirty Page Table

- contain important information for the recovery process
- *transaction table*:
 - 1 entry / active transaction
 - fields: *transID*, *lastLSN* (LSN of the most recent log record for the transaction), *status* (in progress / committed / aborted)
 - example (*status* not displayed):

transID	lastLSN
T10	
T15	

prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB

log

transaction table

The Transaction Table and the Dirty Page Table

- *dirty page table*:
 - 1 entry / dirty page in the Buffer Pool
 - fields: *pageID*, *recLSN* (the LSN of the 1st log record that dirtied the page)

pageID	recLSN
P100	
P2	
P10	

dirty page table

transID	lastLSN
T10	
T15	

transaction table

prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB

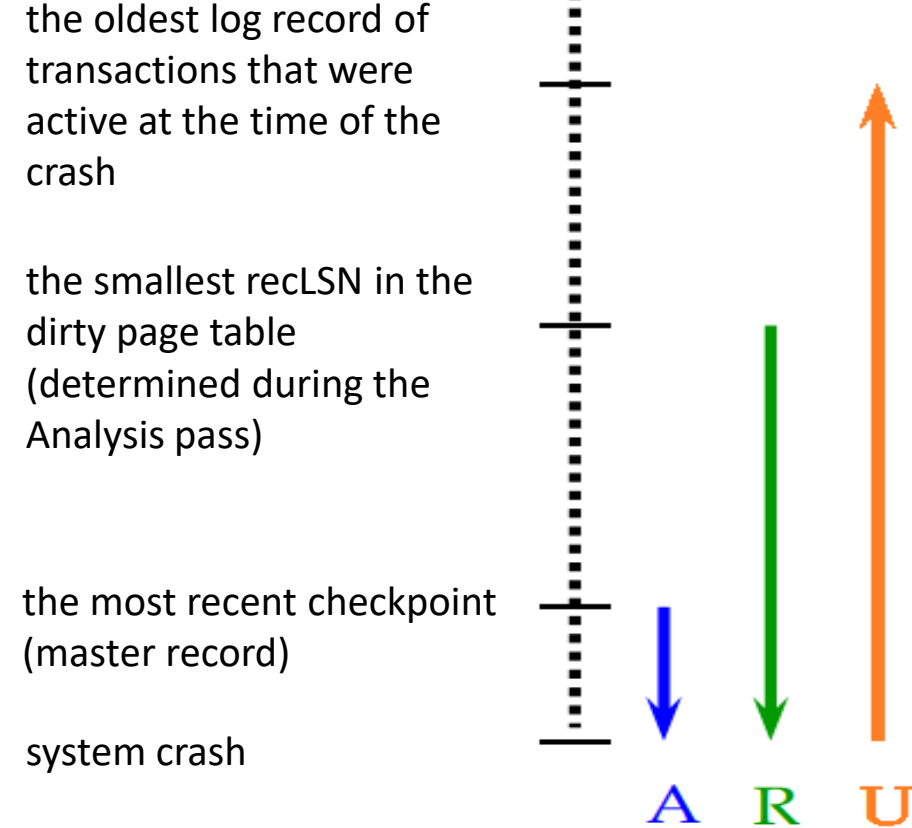
log

Checkpointing

- objective: reduce the amount of work performed by the system when it comes back up after a crash
- *checkpoints* taken periodically; 3 steps:
 - write a *begin_checkpoint* record (it indicates when the checkpoint starts; let its LSN be LSN_{BCK})
 - write an *end_checkpoint* record
 - it includes the current Transaction Table and the current Dirty Page Table
 - after the *end_checkpoint* record is written to stable storage:
 - write a *master* record to a known place on stable storage
 - it includes LSN_{BCK}
- crash -> restart -> system looks for the most recent checkpoint
- normal execution begins with a checkpoint with an empty Transaction Table and an empty Dirty Page Table

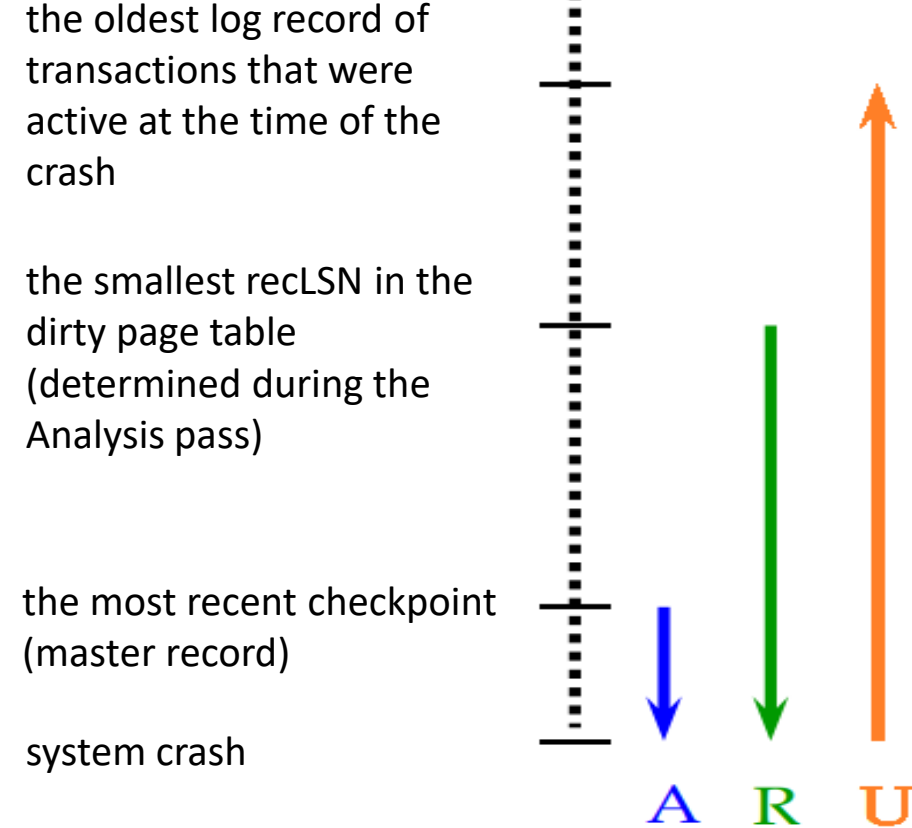
Recovery - overview

- system restart after a crash – 3 phases:
 - **Analysis**
 - reconstructs state at the most recent checkpoint
 - scans the log forward from the most recent checkpoint
 - identifies:
 - active transactions at the time of the crash (to be undone)
 - potentially dirty pages at the time of the crash
 - the starting point for the Redo pass
 - **Redo**
 - repeats history, i.e., reapplies changes to dirty pages



Recovery - overview

- system restart after a crash – 3 phases
 - **Redo**
 - all updates are reapplied (regardless of whether the corresponding transaction committed or not)
 - starting point is determined in the Analysis pass
 - scans the log forward until the last record
 - **Undo**
 - the effects of transactions that were active at the time of the crash are undone
 - such changes are undone in the opposite order (i.e., Undo scans the log backward from the last record)



* Analysis

- investigate the most recent *begin_checkpoint* log record
 - get the next *end_checkpoint* log record EC
- set Dirty Page Table to the copy of the Dirty Page Table in EC
- set Transaction Table to the copy of the Transaction Table in EC

->

* Analysis - scan the log forward from the most recent checkpoint:

- transactions:


- encounter **end log record** for transaction T:
 - remove T from transaction table
- encounter **other log records (LR)** for transaction T:
 - add T to Transaction Table if not already there
 - set T.lastLSN to LR.LSN
 - if LR is a commit type log record:
 - set T's status to *C*
 - otherwise, set status to *U* (i.e., to be undone)

- pages:

- encounter **redoable log record (LR)** for page P:
 - if P is not in the Dirty Page Table:
 - add P to Dirty Page Table
 - set P.recLSN to LR.LSN

Example 1

- first 5 log records are written to stable storage
- system crashes before the 6th log record is written to stable storage



prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB
	T15	commit					
	T10	update	P11	3	20	GFX	YTR

log

Analysis


- most recent checkpoint – beginning of execution (empty Transaction Table, empty Dirty Page Table)
- 1st log record
 - add T10 to the Transaction Table
 - add P100 to the Dirty Page Table (recLSN = LSN(1st log record))

Analysis

- 2nd log record

- add T15 to the Transaction Table
- add P2 to the Dirty

Page Table (recLSN = LSN(2nd log record))



prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB
	T15	commit					

log

- 4th log record

- add P10 to the Dirty Page Table (recLSN = LSN(4th log record))


- active transactions at the time of the crash:

- transactions with status *U*, i.e., T10 (T15 is a committed transaction)

- Dirty Page Table:

- can include pages that were written to disk prior to the crash
- assume P2's update is the only change written to disk before the crash, i.e., P2 is not dirty, but it's in the Dirty Page Table
- the pageLSN on page P2 is equal to the LSN of the 2nd log record

Analysis



prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB
	T15	commit					

log

- log record

T10	update	P11	3	20	GFX	YTR
-----	--------	-----	---	----	-----	-----

 is not seen during Analysis (it was not written to disk before the crash)
- Write-Ahead Logging protocol => the corresponding change to page P11 cannot have been written to disk

* Redo

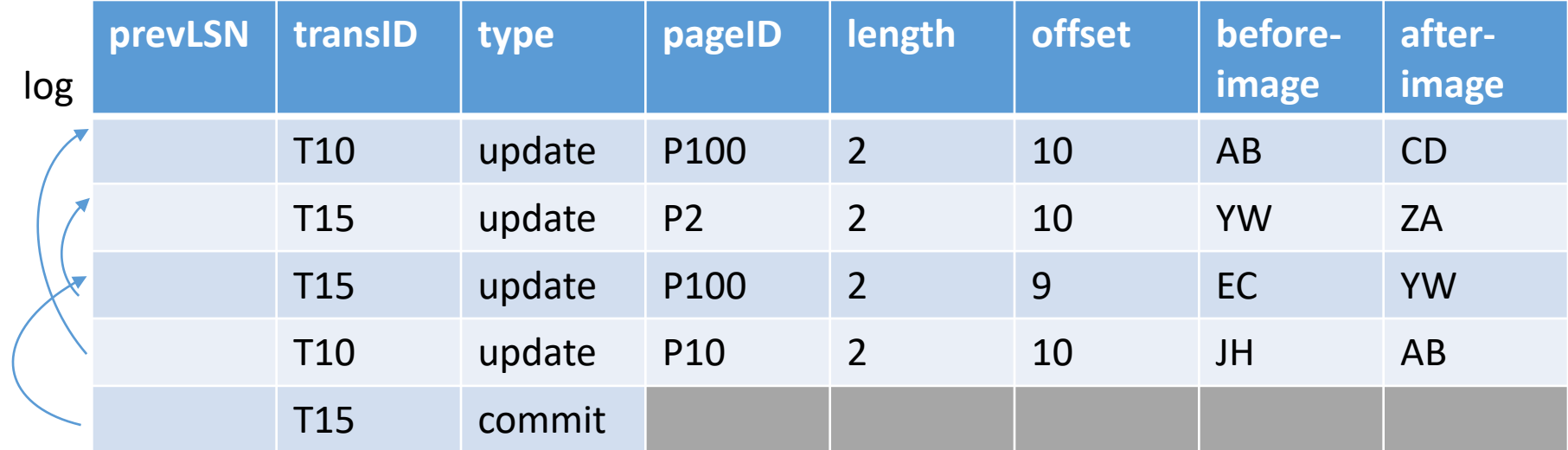
- *repeat history*: reconstruct state at the time of the crash
 - reapply *all* updates (even those of aborted transactions!), reapply CLR's
- scan the log forward from the log record with the smallest recLSN in the Dirty Page Table
- for each **redoable** log record **LR** affecting page P, redo the described action unless:
 - page P is not in the Dirty Page Table
 - page P is in the Dirty Page Table, but $P.\text{recLSN} > LR.\text{LSN}$
 - $P.\text{pageLSN (in DB)} \geq LR.\text{LSN}$
- to redo an action:
 - reapply the logged action
 - set $P.\text{pageLSN}$ to $LR.\text{LSN}$
 - no additional logging!

* Redo

- at the end of Redo:
 - for every transaction T with status C:
 - add an end log record
 - remove T from the Transaction Table

Redo

- previously stated assumption: P2's update is the only change written to



	prevLSN	transID	type	pageID	length	offset	before-image	after-image
		T10	update	P100	2	10	AB	CD
		T15	update	P2	2	10	YW	ZA
		T15	update	P100	2	9	EC	YW
		T10	update	P10	2	10	JH	AB
		T15	commit					

disk before the crash, i.e., P2 is not dirty, but it's in the Dirty Page Table

- Dirty Page Table -> smallest recLSN is the LSN of the 1st log record
- 1st log record
 - fetch page P100 (its pageLSN is less than the LSN of the current log record) => reapply update, set P100.pageLSN to the LSN of the 1st log record
- 2nd log record
 - fetch page P2
 - P2.pageLSN = LSN of the current log record => update is not reapplied
- 3rd, 4th log records – processed similarly

* Undo

- *loser transaction* – transaction that was active at the time of the crash
 - ToUndo = { l | l - lastLSN of a *loser* transaction }
 - repeat:
 - choose the largest LSN in ToUndo and process the corresponding log record LR; let T be the corresponding transaction
 - if LR is a CLR:
 - if undoNextLSN == NULL
 - write an end log record for T
 - else {undoNextLSN != NULL}
 - add undoNextLSN to ToUndo
 - else {LR is an update log record}
 - undo the update
 - write a CLR
 - add LR.prevLSN to ToUndo
- until ToUndo is empty

Undo

- active transaction at the time of the crash: T10

- lastLSN of T10: LSN of the 4th log record


- 4th log record

- undo update, write CLR
- add LSN of 1st log record to ToUndo

- 1st log record

- undo update (!T15's change to P100 is lost!)
- write CLR
- write end log record for T10

- obs. if Strict 2PL is used, T15 cannot write P100 while T10 is active (T10 has also modified P100)

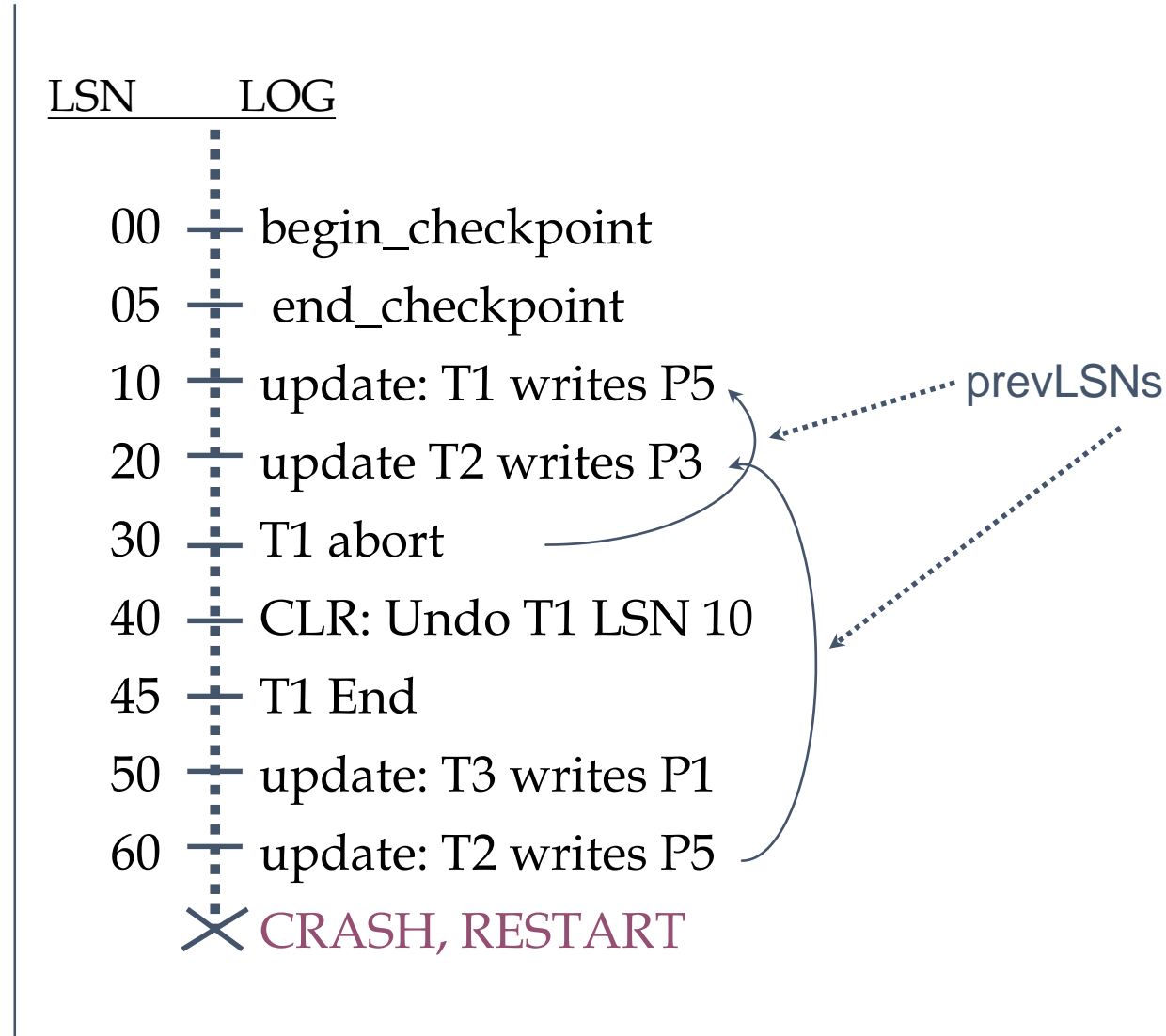


prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB
	T15	commit					

log

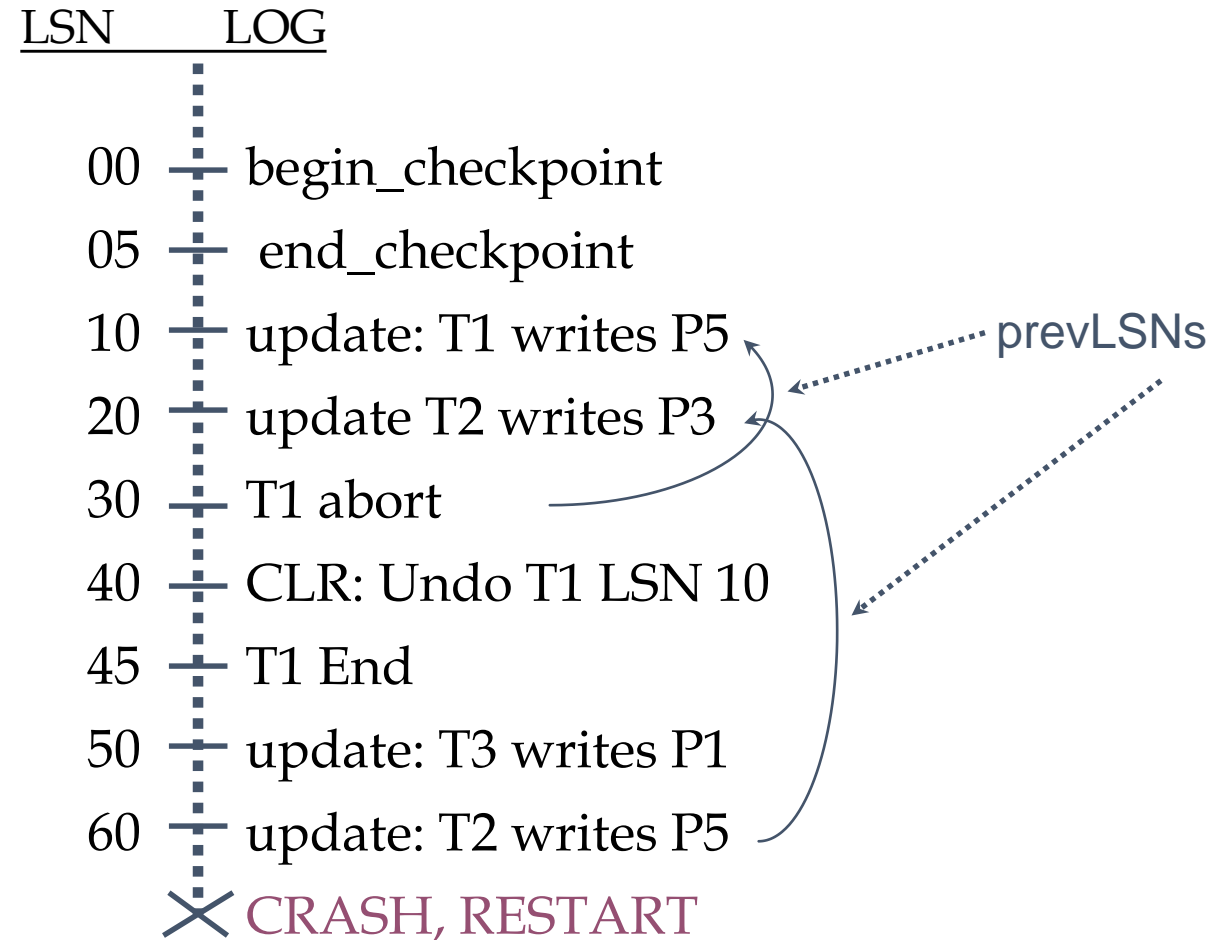
Example 2 – system crashes during Undo

- consider the execution history below:



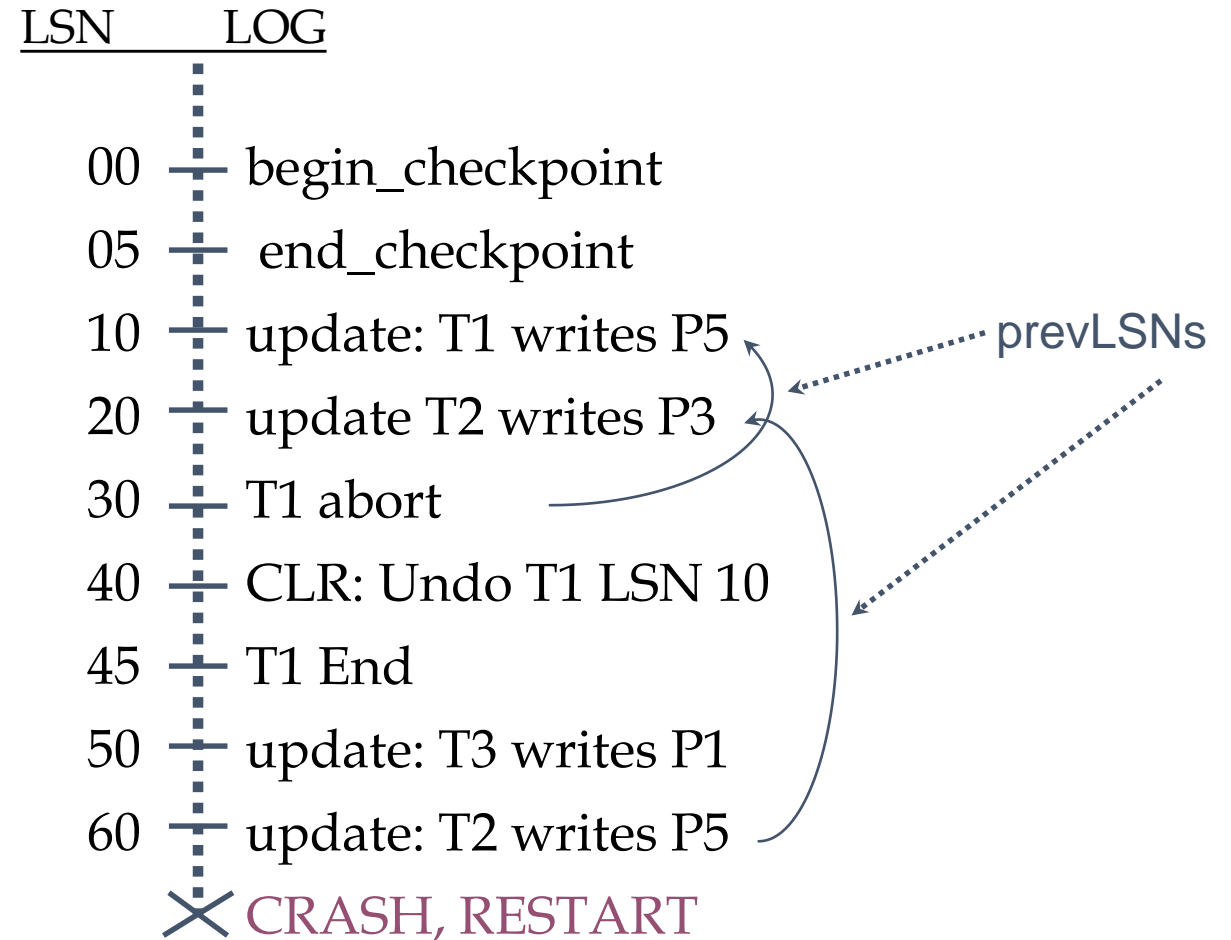
Example 2

- T1 aborts
=> its only update is undone
(CLR with LSN 40)
- T1 - terminated
- **1st crash:**
 - **Analysis:**
 - dirty pages: P5 (recLSN 10), P3 (recLSN 20), P1 (recLSN 50)
 - active transactions at the time of the crash: T2 (lastLSN 60), T3 (lastLSN 50)



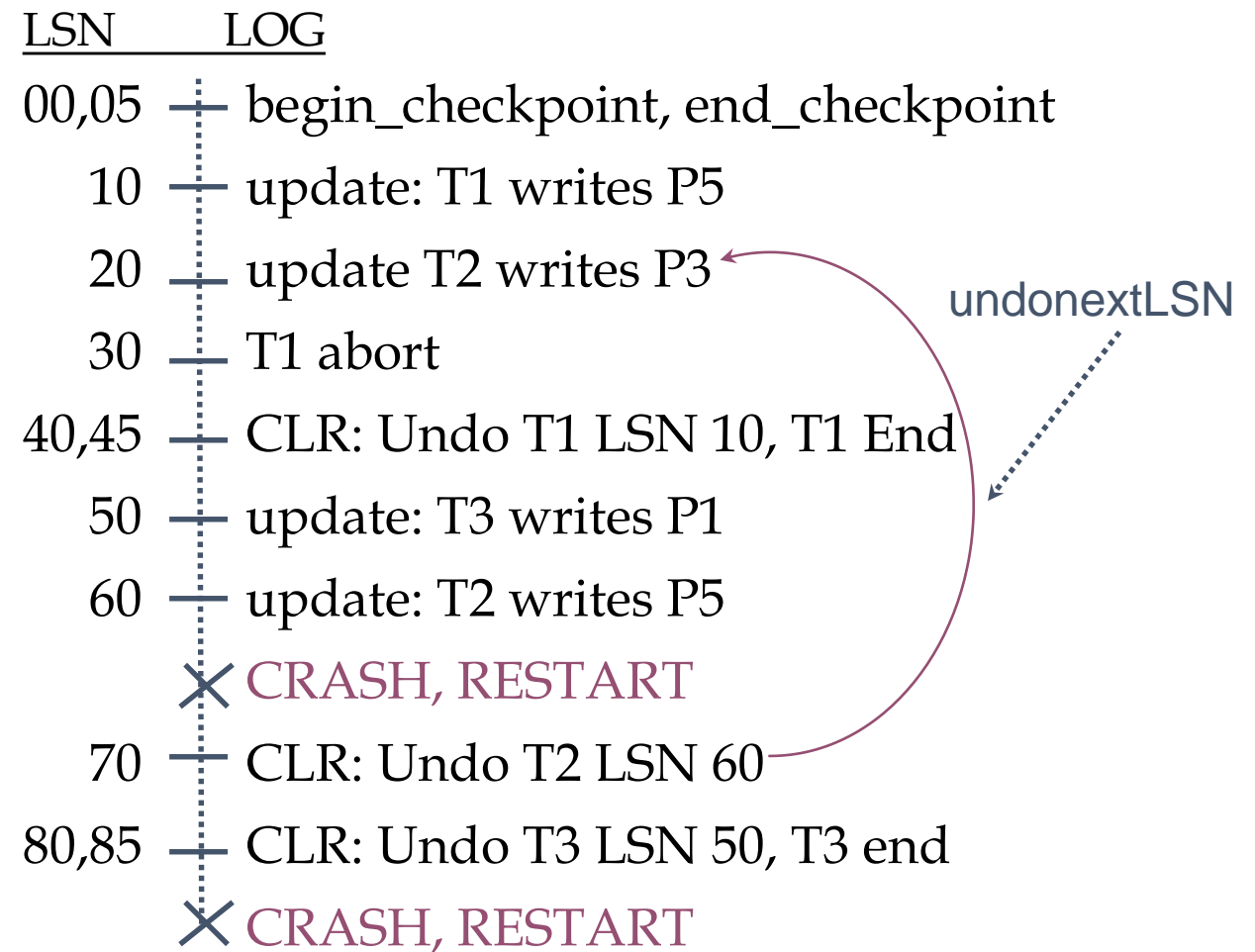
Example 2

- **1st crash:**
 - **Redo:**
 - starting point
 - log record with LSN = 10 (smallest recLSN in the Dirty Page Table)
 - reapply required actions in update log records / compensation log records



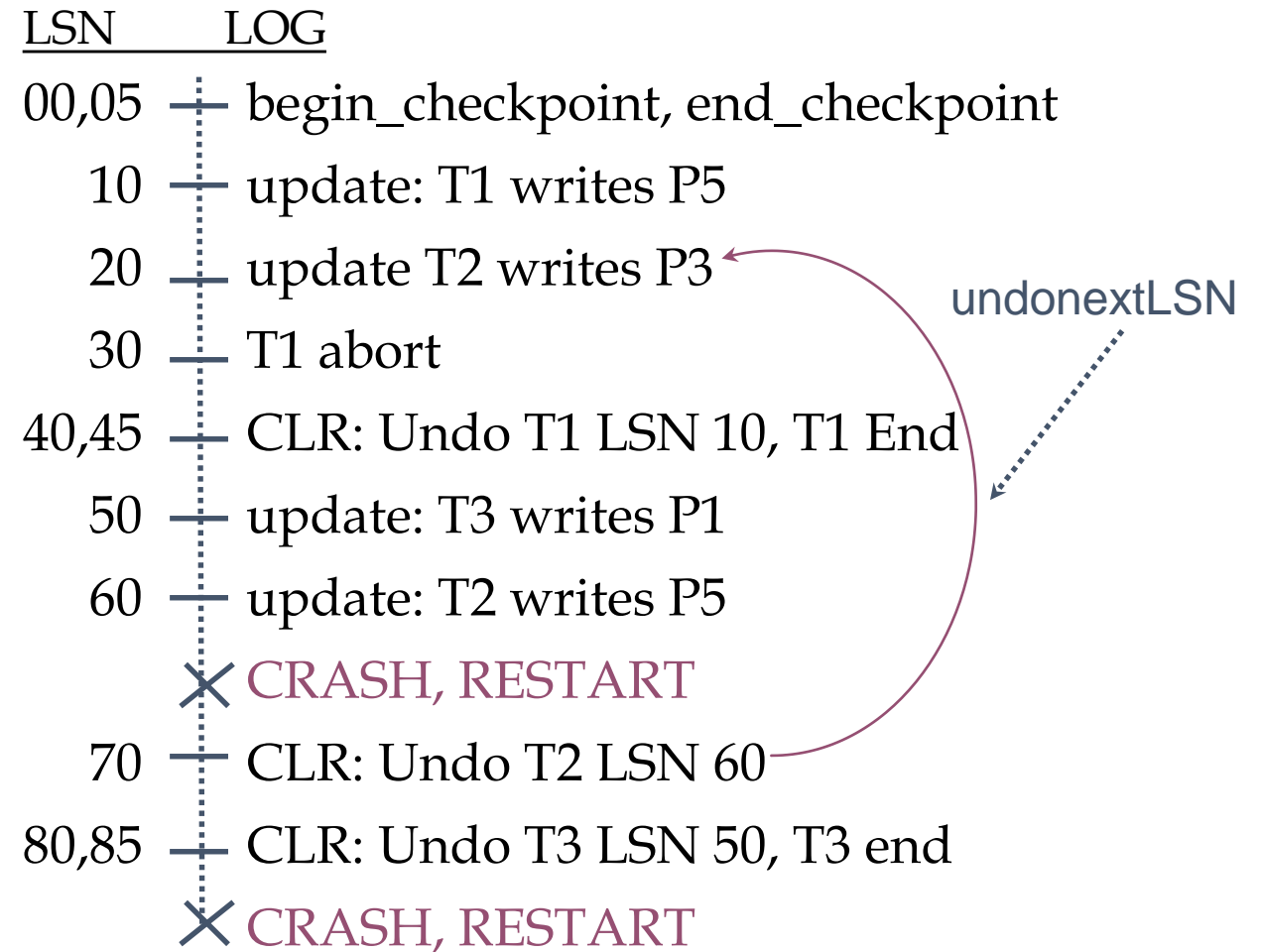
Example 2

- **1st crash:**
 - **Undo:**
 - T2, T3 – loser transactions
=> ToUndo = {60, 50}
 - process log record with LSN 60:
 - undo update
 - write CLR (LSN 70) with undoNextLSN 20 (i.e., the next log record that should be processed for T2)



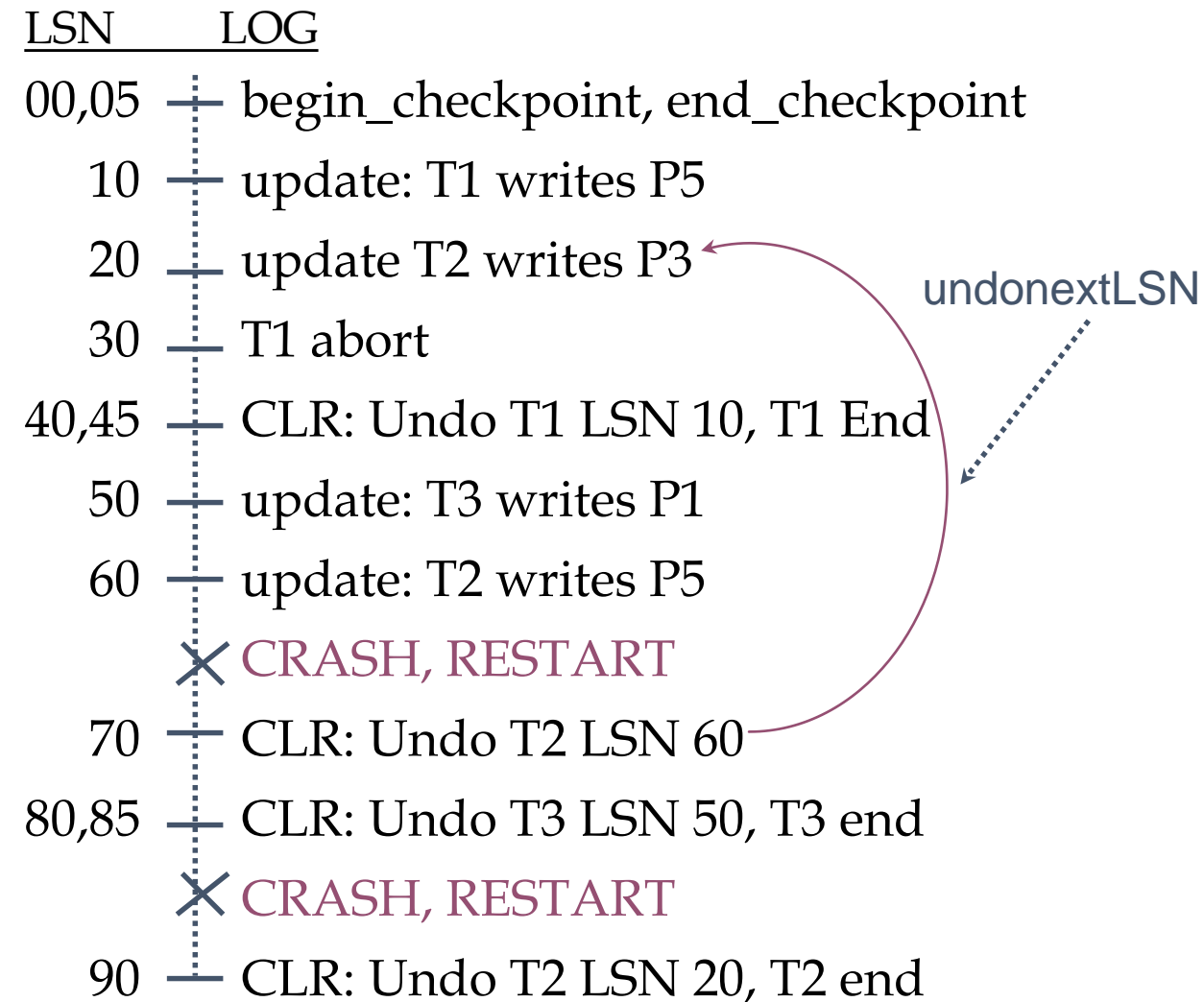
Example 2

- **1st crash:**
 - **Undo:**
 - process log record with LSN 50:
 - undo update
 - write CLR (LSN 80) with undoNextLSN *null* (i.e., T3 completely undone, write end log record for T3)
 - log records with LSN 70, 80, 85 are written to stable storage
- **2nd crash (during undo)!**



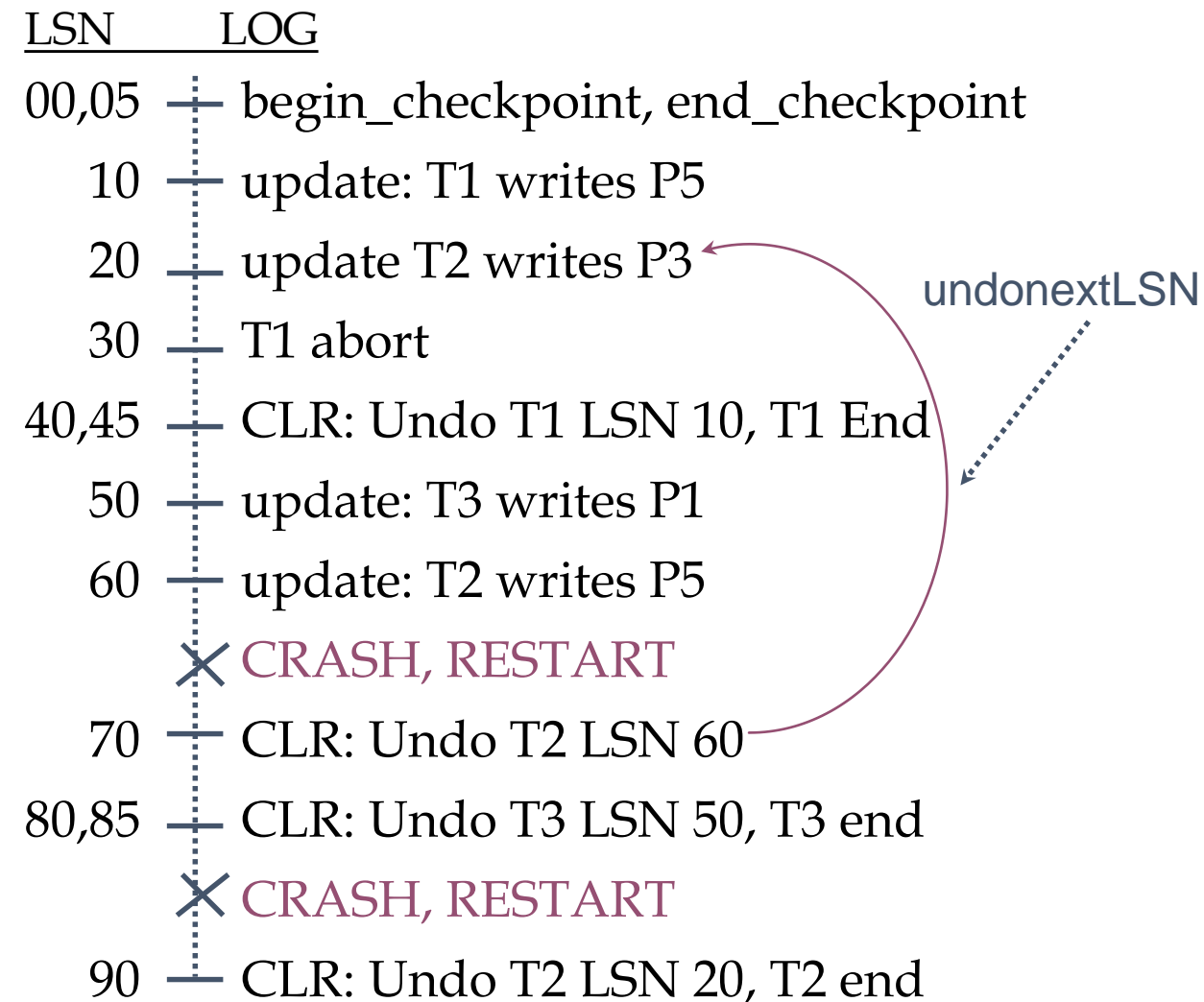
Example 2

- **2nd crash:**
 - **Analysis:**
 - the only active transaction: T2
 - dirty pages: P5 (recLSN 10), P3 (recLSN 20), P1 (recLSN 50)
 - **Redo:**
 - process log records with LSN between 10 and 85



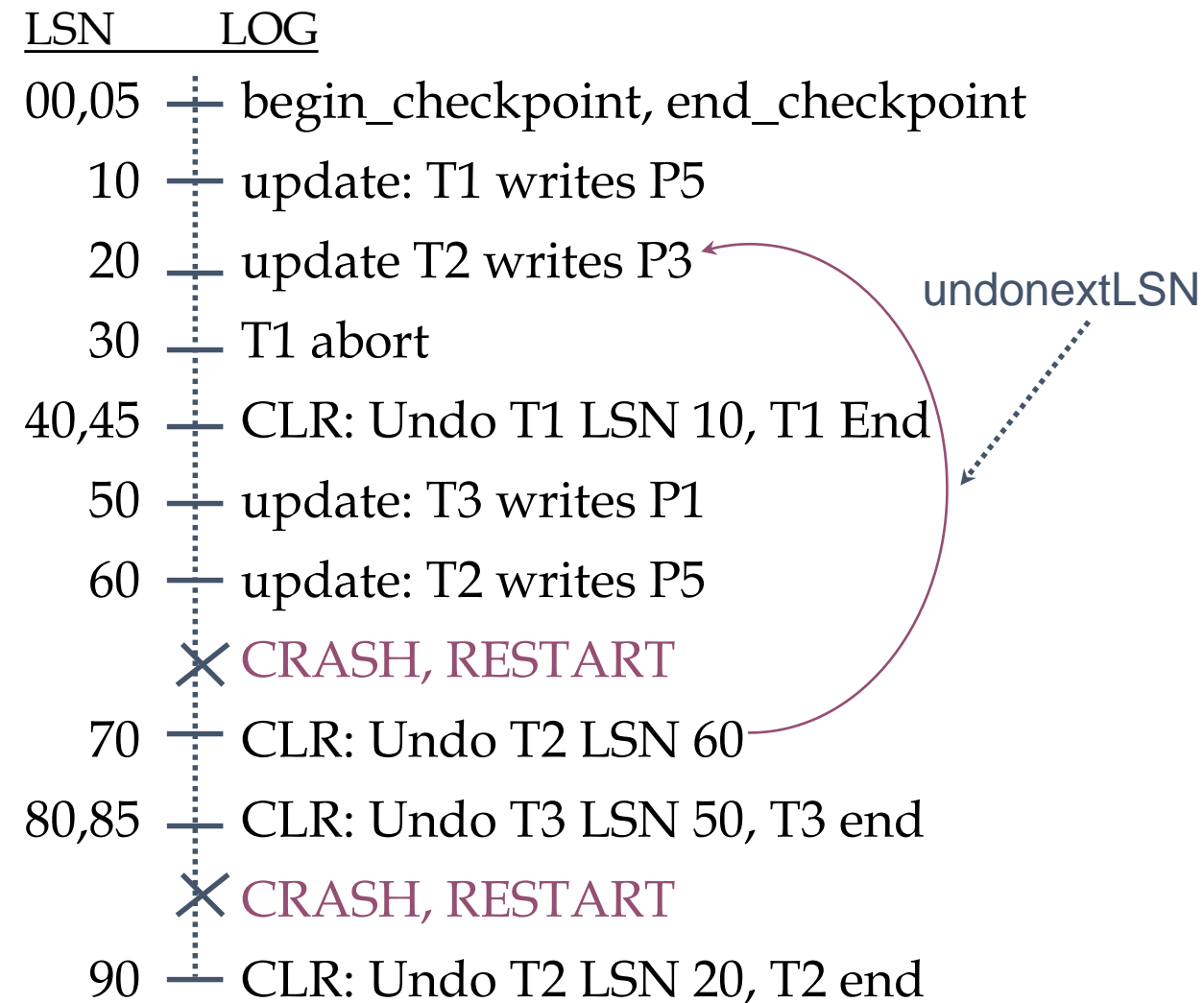
Example 2

- 2nd crash:
 - Undo:
 - lastLSN of T2: 70
 - ToUndo = {70}
 - process log record with LSN 70:
 - add 20 (undoNextLSN) to ToUndo
 - process log record with LSN 20:
 - undo update
 - write CLR (LSN 90) with undoNextLSN *null* => write end log record for T2



Example 2

- 2nd crash:
 - Undo:
 - ToUndo empty
- => recovery complete!



- obs. aborting a transaction
 - special case of Undo in which the actions of a single transaction are undone
- obs. system crash during the Analysis pass
 - all the work is lost
 - when the system comes back up, the Analysis phase has the same information as before
- obs. system crash during the Redo pass
 - some of the changes from the Redo pass may have been written to disk prior to the crash
 - the pageLSN will indicate such a situation, so these changes will not be reapplied in the subsequent Redo pass

References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2nd Edition), McGraw-Hill, 2000
- [Le99] LEVENE, M., LOIZOU, G., A Guided Tour of Relational Databases and Beyond, Springer, 1999
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007,
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, <http://codex.cs.yale.edu/avi/db-book/>