ARIES: A Transaction Recovery Method

Presentation: Dorna
Discussion: Tanya

*Some slides borrowed from Rachel's notes, and CMU DB course

ACID properties

- Atomicity: Either all actions in the Xact occur, or none occur.
- Consistency: If each Xact is consistent, and the DB starts in a consistent state, then the DB ends up being consistent.
- Isolation: The execution of one Xact is isolated from that of other Xacts.
- Durability: If a Xact commits, then its effects persist.

What happens if the system fails?

- The goal of transaction recovery is to resurrect the db if this happens
- ARIES is one example of such a system
- Write-Ahead Logging:
  - Any change is recorded in log on stable storage before the database change is written to disk
  - Must use STEAL + NO-FORCE buffer pool policies
- Repeating History During Redo:
  - On DBMS restart, retrace actions and restore database to exact state before crash
- Logging Changes During Undo:
  - Record undo actions to log to ensure action is not repeated in the event of repeated failures

Discussion 1 (Groups of 4)

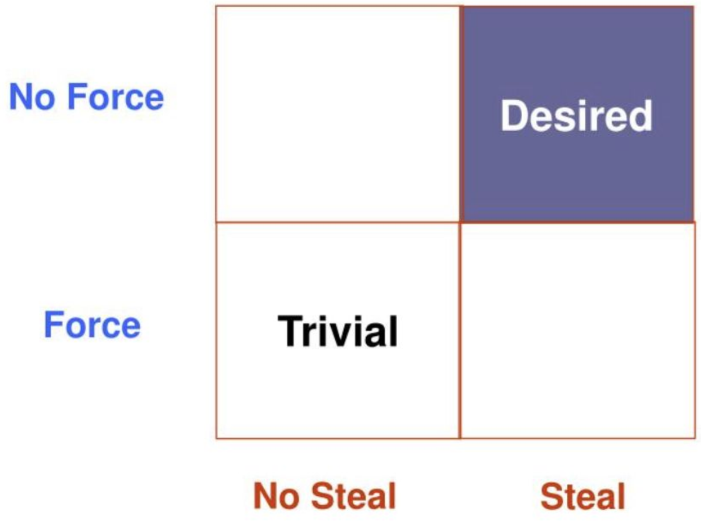
- With different buffer pool strategies (STEAL/NO-STEAL, FORCE/NO-FORCE), the designer is engaging in a trade-off. Which one is more important(Runtime performance vs. Recovery performance)? Why/When?
- With the increasing speed, storage capacity, bandwidth, and reliability of modern storage systems, how does this impact the relative cost and overhead of log records?

9 goals of ARIES:

- Simplicity
- Operation Logging
  - “let one transaction modify the same data that was modified earlier by another transaction which has not yet committed, when the two transactions’ actions are semantically compatible”
- Flexible storage management
- Partial rollbacks
  - Support save points and rollbacks to save points in order to be user friendly
- Flexible buffer management
  - Make the least number of restrictive assumptions about buffer management policies
- Recovery independence
  - “The recovery of one object should not force the concurrent or lock-step recovery of another object”
- Logical undo
- Parallelism and fast recovery
- Minimal overhead

Handling the buffer pool

- Transactions modify pages in memory buffers
- Writing to disk is more permanent
- When should updated pages be written to disk?
- Force every write to disk
    - Poor response time
    - But provides durability
- Steal buffer-pool frames from uncommitted Xacts (resulting in write to disk)
    - If not, poor throughput
    - If so, how can we ensure atomicity?



Logging

- Record ordered REDO and UNDO information, for every update, in a log
- Sequential writes to log (put it on a separate disk)
- Minimal info (diff) written to log, so multiple updates fit in a single log page
- Log record contains:
  - <XID, pageID, offset, length, old data, new data>
  - May have other fields
- Each log record has a unique Log Sequence Number (LSN)
  - LSNs always increasing

LogRecord fields:

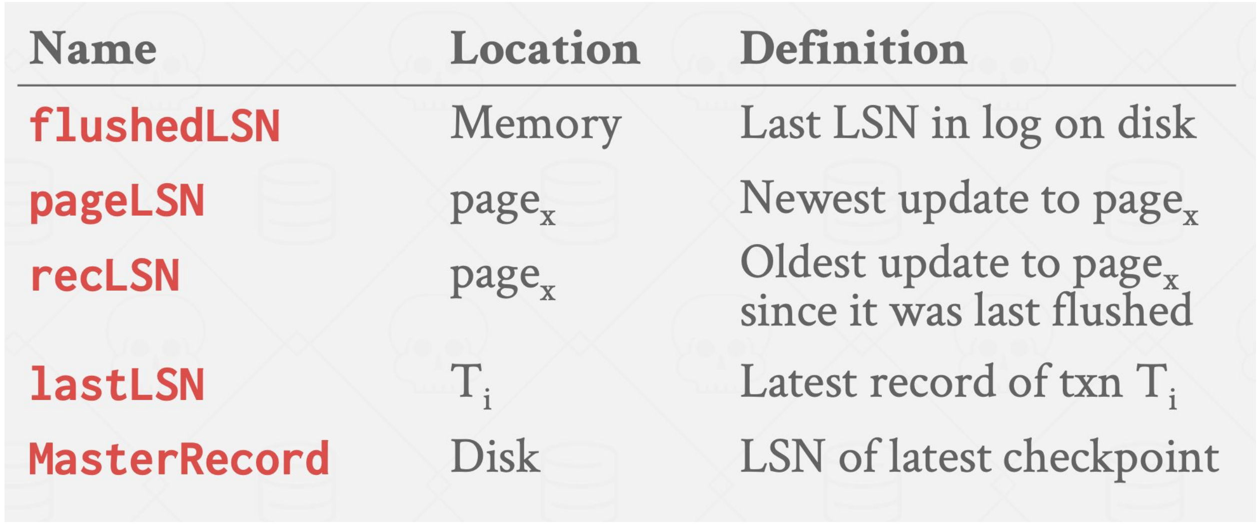
prevLSN
transID
type

update records only $\left\{ \begin{array}{l} \text{pageID} \\ \text{length} \\ \text{offset} \\ \text{before-image} \\ \text{after-image} \end{array} \right.$

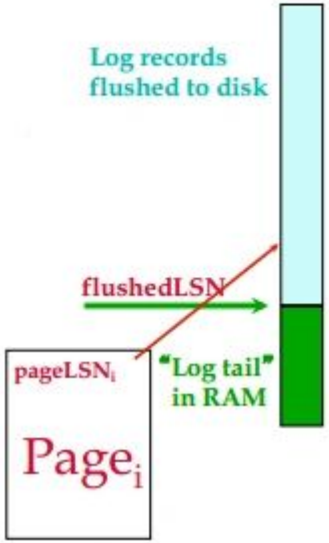
Write-ahead logging

- Must force log record for an update before the corresponding data page gets to disk
  - guarantees Atomicity
- Must write all log records for a Xact before commit
  - guarantees Durability

LSN



Name	Location	Definition
flushedLSN	Memory	Last LSN in log on disk
pageLSN	page_x	Newest update to page_x
recLSN	page_x	Oldest update to page_x since it was last flushed
lastLSN	T_i	Latest record of txn T_i
MasterRecord	Disk	LSN of latest checkpoint



- Before the DBMS can write page x to disk, it must flush the log at least to the point where:
  - pageLSNx ≤ flushedLSN
  - I.e., the latest thing on disk mustalso be written to disk on the log

Log record types

- Update
  - Inserted when modifying a page
  - Contains all the fields
  - pageLSN of that page is set to the LSN of the record (i.e., page updated)
- Commit
  - When Xact commits a record is written in the log and is forcibly written to stable storage
- Abort
  - created when Xact is aborted
- End (signifies end of commit or abort)
  - created when Xact has completed all work (after commit or abort)
- Compensation Log Records (CLRs) for UNDO actions
  - Inserted before undoing an action described by an update log record
  - It happens during aborting or recovery
  - Contains undoNextLSN field: LSN of next log record to be undone

Discussion 2 - Groups of 2

- The authors claim that the system is simple and efficient. Why or why not?
- Discuss what you found hard to understand, something that was too complex.
- If you found it very intuitive and simple, discuss its simplicity.

Tables

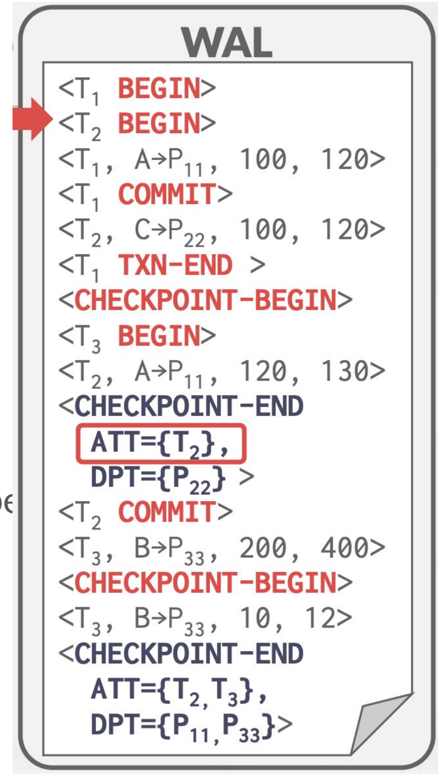
- Active Transaction Table (ATT):
    - Maintained by transaction manager
    - Has one entry per active Xact
    - Contains <tranID, status (running/committed/undo), and lastLSN (LSN of most recent log record for it)>
    - Xact removed from table when end record is inserted in the log

Tables (cont.)

- Dirty Page Table (DPT):
  - Maintained by buffer manager
  - Has one entry per dirty page in buffer pool
  - Contains recLSN -- LSN of action which first made the page dirty
  - Entry is removed when page is written to the disk
- Both tables must be reconstructed during recovery

Checkpoints

- minimize recovery time in system crash
  - Periodically fuzzy checkpoint
  - begin_checkpoint record: when checkpoint began
  - end_checkpoint record: current ATT and DPT
  - Xacts continue to run; so these tables are accurate only as of time of be
  - Dirty pages are not forced to disk
  - Store LSN of checkpoint record in a safe place (MasterRecord)
- When system starts after a crash:
  - Locate the most recent checkpoint
  - Restore Xact table and dirty page table from there



```
<T1 BEGIN>
<T2 BEGIN>
<T1, A→P11, 100, 120>
<T1 COMMIT>
<T2, C→P22, 100, 120>
<T1 TXN-END >
<CHECKPOINT-BEGIN>
<T3 BEGIN>
<T2, A→P11, 120, 130>
<CHECKPOINT-END
  ATT={T2},
  DPT={P22} >
<T2 COMMIT>
<T3, B→P33, 200, 400>
<CHECKPOINT-BEGIN>
<T3, B→P33, 10, 12>
<CHECKPOINT-END
  ATT={T2,T3},
  DPT={P11,P33}>
```

Discussion 3 - Groups of 3

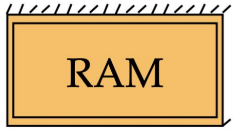
- Do you think that fuzzy checkpoints are a good idea? Would you use them? Why or why not? Does it depend on the circumstances?
- What are the advantages and disadvantages of using fuzzy checkpoints instead of using something more simple?

What's stored where

LogRecords
- prevLSN
- XID
- type
- pageID
- length
- offset
- before-image
- after-image

Data pages
each with a pageLSN

master record
LSN of most recent checkpoint



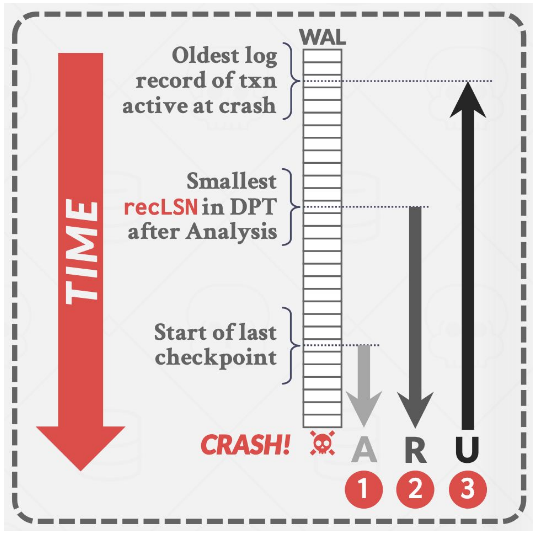
Xact Table
- lastLSN
- status

Dirty Page Table
- recLSN

flushedLSN

Crash recovery

- Start from a checkpoint (found via master record)
- Three phases. Need to:
  - Figure out which Xacts committed since checkpoint, which failed (Analysis)
  - REDO all actions (repeat history)
  - UNDO effects of failed Xacts


WAL
Oldest log record of txn active at crash
Smallest recLSN in DPT after Analysis
Start of last checkpoint
TIME
CRASH!
A
R
U
1
2
3

Recovery: Analysis

- Goals:
  - Determine log record that Redo has to start at
  - Determine pages that were dirty at crash
  - Identify Xact’s active at crash
- Reconstruct state at checkpoint
  - reconstruct ATT & DPT using end_checkpoint record
- Scan log forward from checkpoint
  - End record: Remove Xact from Xact table
  - Other bookkeeping happens

Recovery: REDO

- repeat history to reconstruct state at crash:
  - Reapply all updates (even of aborted Xacts), redo CLRs
- Scan forward from log record containing smallest recLSN in DPT
  - For each CLR or update log record, REDO the action (unless it's clear that it's already been recorded)
- To REDO an action:
  - Reapply logged action
  - Set pageLSN to LSN
  - No additional logging is required
- At the end of REDO, and End record is inserted in the log for each transaction with status C which is removed from ATT

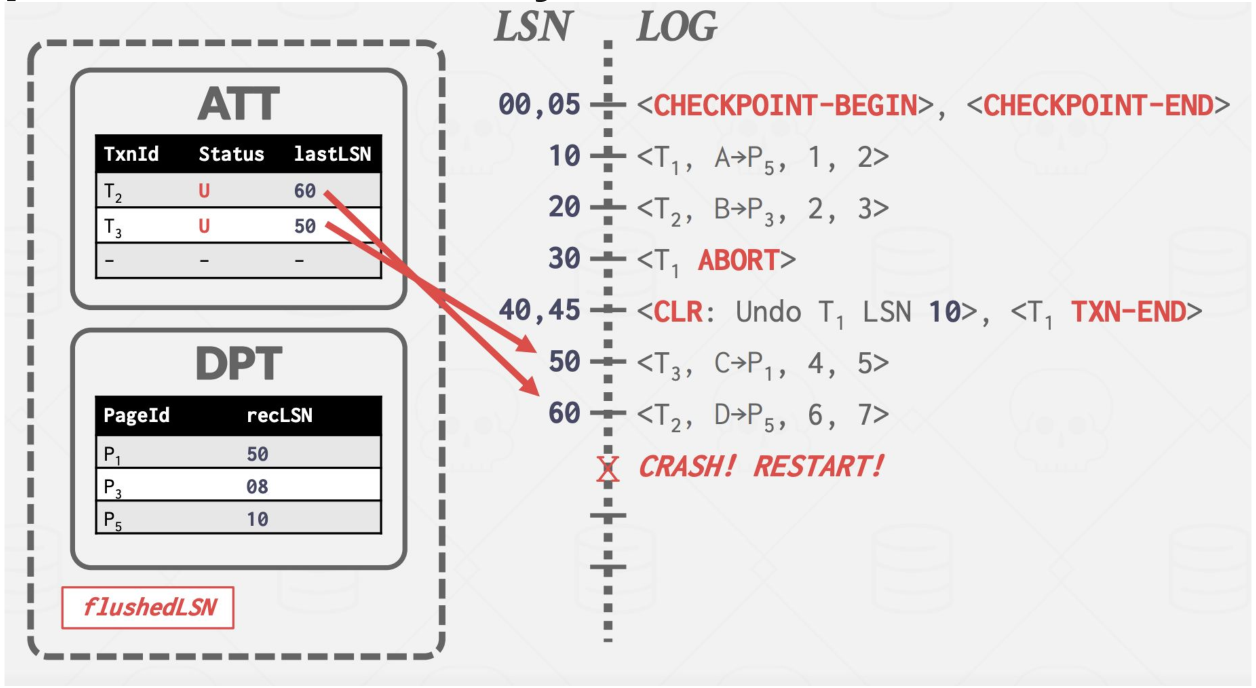
Recovery: Undo

- Xact active at the crash: losers!
- undo all records of loser Xact’s in reverse order
    - ToUndo: set of all lastLSN values of all loser Xact’s

Recovery: Undo (Cont.)

- Repeat until ToUndo is empty:
    - Choose largest LSN among ToUndo
    - If this LSN is a CLR and undonextLSN==NULL
        - write an End record for this Xact.
        - remove record from ToUndo set
    - If this LSN is a CLR, and undonextLSN != NULL
        - add undonextLSN to ToUndo
    - Else this LSN is an update
        - undo the update, write a CLR
        - remove record from toUndo
        - add prevLSN of this record to ToUndo

Example of full recovery



ATT

TxnId	Status	lastLSN
T_2	U	60
T_3	U	50
-	-	-

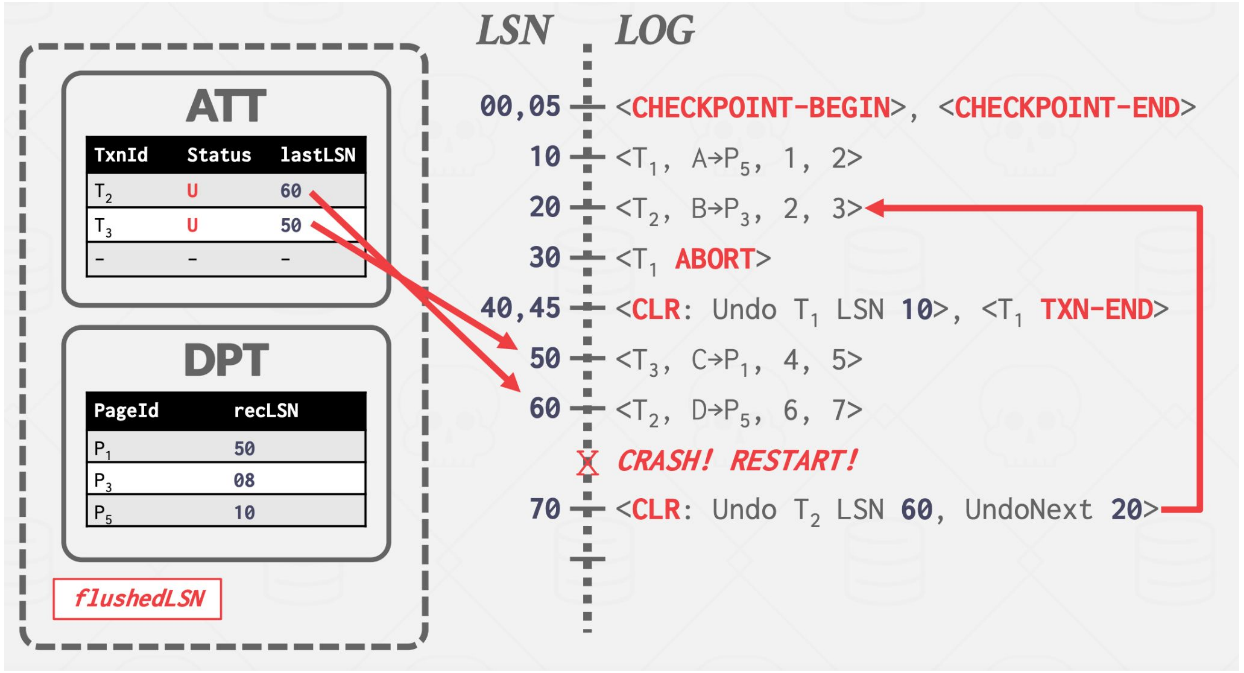
DPT

PageId	recLSN
P_1	50
P_3	08
P_5	10

flushedLSN

LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1 ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1 TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
	CRASH! RESTART!

Example of full recovery



ATT

TxnId	Status	lastLSN
T_2	U	60
T_3	U	50
-	-	-

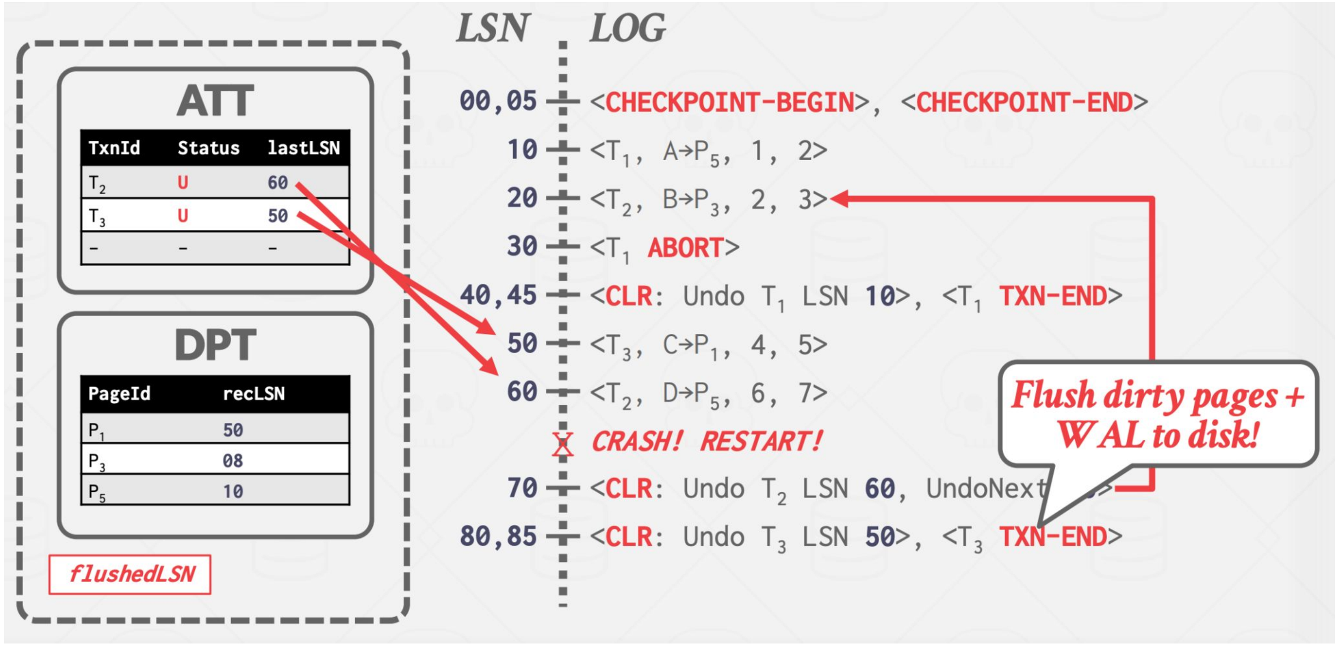
DPT

PageId	recLSN
P_1	50
P_3	08
P_5	10

flushedLSN

LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1 ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1 TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
	CRASH! RESTART!
70	<CLR: Undo T_2 LSN 60, UndoNext 20>

Example of full recovery



ATT

TxnId	Status	lastLSN
T_2	U	60
T_3	U	50
–	–	–

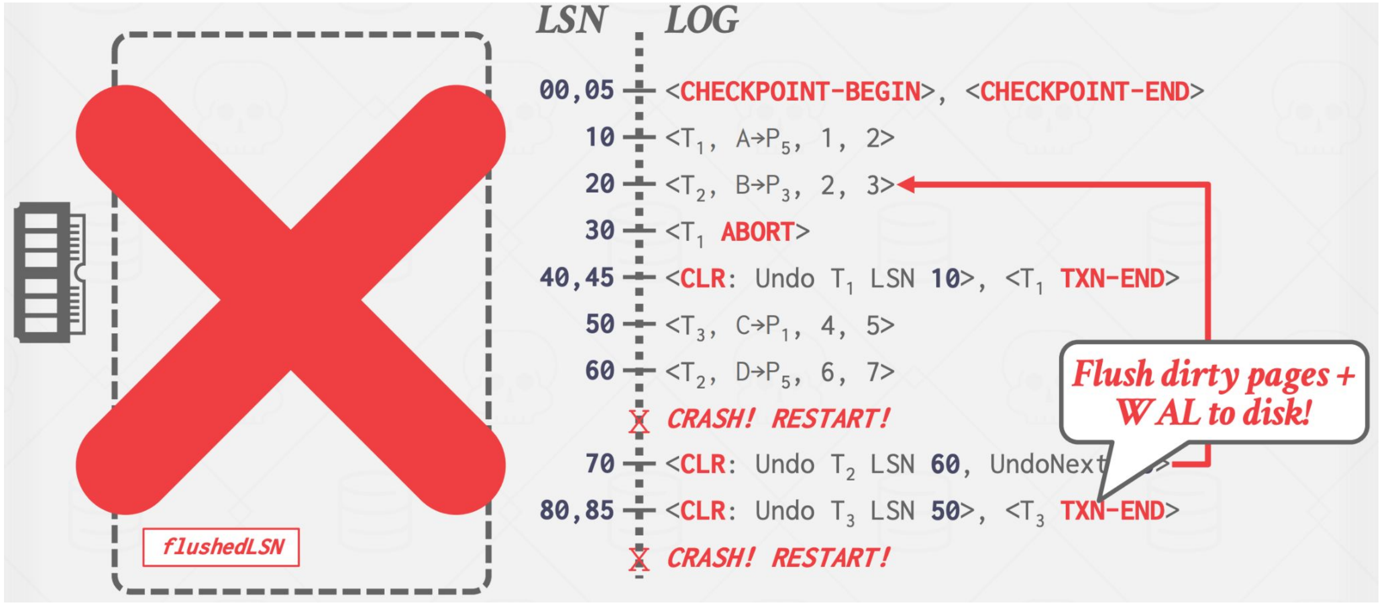
DPT

PageId	recLSN
P_1	50
P_3	08
P_5	10

flushedLSN

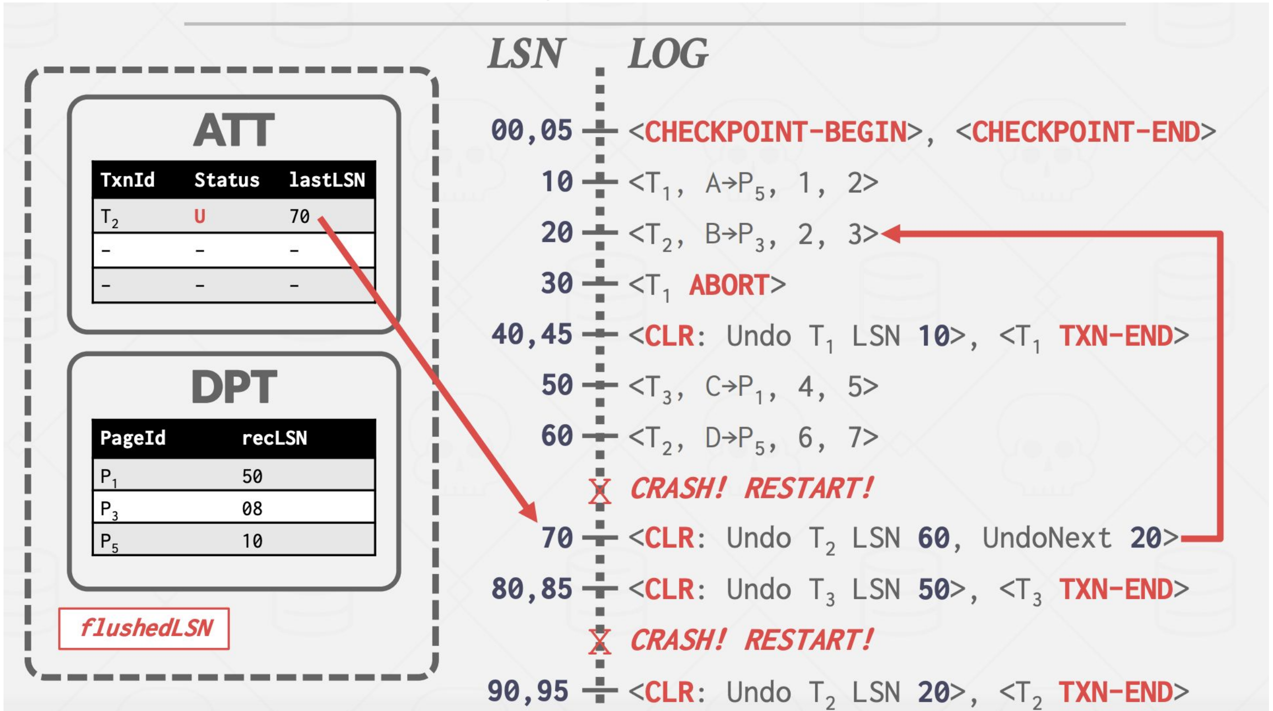
LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1 ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1 TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
	CRASH! RESTART!
70	<CLR: Undo T_2 LSN 60, UndoNext
80,85	<CLR: Undo T_3 LSN 50>, <T_3 TXN-END>

Example of full recovery



```
LSN     LOG
00,05   <CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10      <T1, A→P5, 1, 2>
20      <T2, B→P3, 2, 3>
30      <T1 ABORT>
40,45   <CLR: Undo T1 LSN 10>, <T1 TXN-END>
50      <T3, C→P1, 4, 5>
60      <T2, D→P5, 6, 7>
        CRASH! RESTART!
70      <CLR: Undo T2 LSN 60, UndoNext
80,85   <CLR: Undo T3 LSN 50>, <T3 TXN-END>
        CRASH! RESTART!
```

Example of full recovery



ATT

TxnId	Status	lastLSN
T_2	U	70
–	–	–
–	–	–

DPT

PageId	recLSN
P_1	50
P_3	08
P_5	10

LSN | LOG

LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1 ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1 TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
	CRASH! RESTART!
70	<CLR: Undo T_2 LSN 60, UndoNext 20>
80,85	<CLR: Undo T_3 LSN 50>, <T_3 TXN-END>
	CRASH! RESTART!
90,95	<CLR: Undo T_2 LSN 20>, <T_2 TXN-END>

Discussion 4 (Groups of 4)

- What properties of ARIES can be an advantage in using it in parallel databases or distributed databases?
- What extensions are needed in ARIES for supporting crash recovery in distributed or parallel databases?

  (For example, consider a database shared between 2 servers, and one of the servers crashes during a transaction (but the other server is functional and is carrying out a transaction of its own). How would crash recovery be ensured in this scenario (how would the locks be freed if they are across both the machines))

Modern recovery algorithms

- Like ARIES
  - maintain a log
  - use WAL
- Different REDO
  - they don’t repeat the whole history
  - they only redo the non-loser transactions – “selective redo”
    - Can lead to trouble because must log undos (for media recovery), then would attempt to redo undo

Discussion 5 (Groups of 2)

- We mostly talk and read about the best work that eventually turned out to be the state-of-the-art method. Should we as researchers pay more attention to the research that failed and why it was not successful? What would be an efficient way to do that? Maybe some old methods are now more applicable with changing requirements.
- Old ideas do tend to come back :)

Thank You!