# CSE 544
# Principles of Database Management Systems

Alvin Cheung

Fall 2015

Lecture 13 - Transactions: recovery

# Announcements

- HW3 is due next Thursday
- Next: Focus on your projects! Only 5 weeks left
  - Make a detailed plan of what you want to accomplish each week
  - **Milestone reports are due next week (see website for instructions)**
  - Poster presentations on Tuesday Dec 15$^{th}$
  - Final reports due on Friday Dec18$^{th}$

# References

- **Concurrency control and recovery.**

  Michael J. Franklin. The handbook of computer science and engineering. A. Tucker ed. 1997

- **Database management systems.**

  Ramakrishnan and Gehrke.

  Third Ed. **Chapters 16 and 18.**

# Outline

- **Review of ACID properties**
  - Today we will cover techniques for ensuring atomicity and durability in face of failures
- **Review of buffer manager and its policies**
- **Write-ahead log**
- **ARIES method for failure recovery**

# ACID Properties

- Atomicity: Either all changes performed by transaction occur or none occurs
- Consistency: A transaction as a whole does not violate integrity constraints
- Isolation: Transactions appear to execute one after the other in sequence
- Durability: If a transaction commits, its changes will survive failures

# What Could Go Wrong?

- Concurrent operations
  - That's what we discussed last time (atomicity and isolation properties)
- Failures can occur at any time
  - Today (isolation and durability properties)

# Problem Illustration

```
Client 1:
    START TRANSACTION
    INSERT INTO SmallProduct(name, price)
        SELECT pname, price
        FROM Product
        WHERE price <= 0.99

    DELETE Product
        WHERE price <=0.99
    COMMIT
```

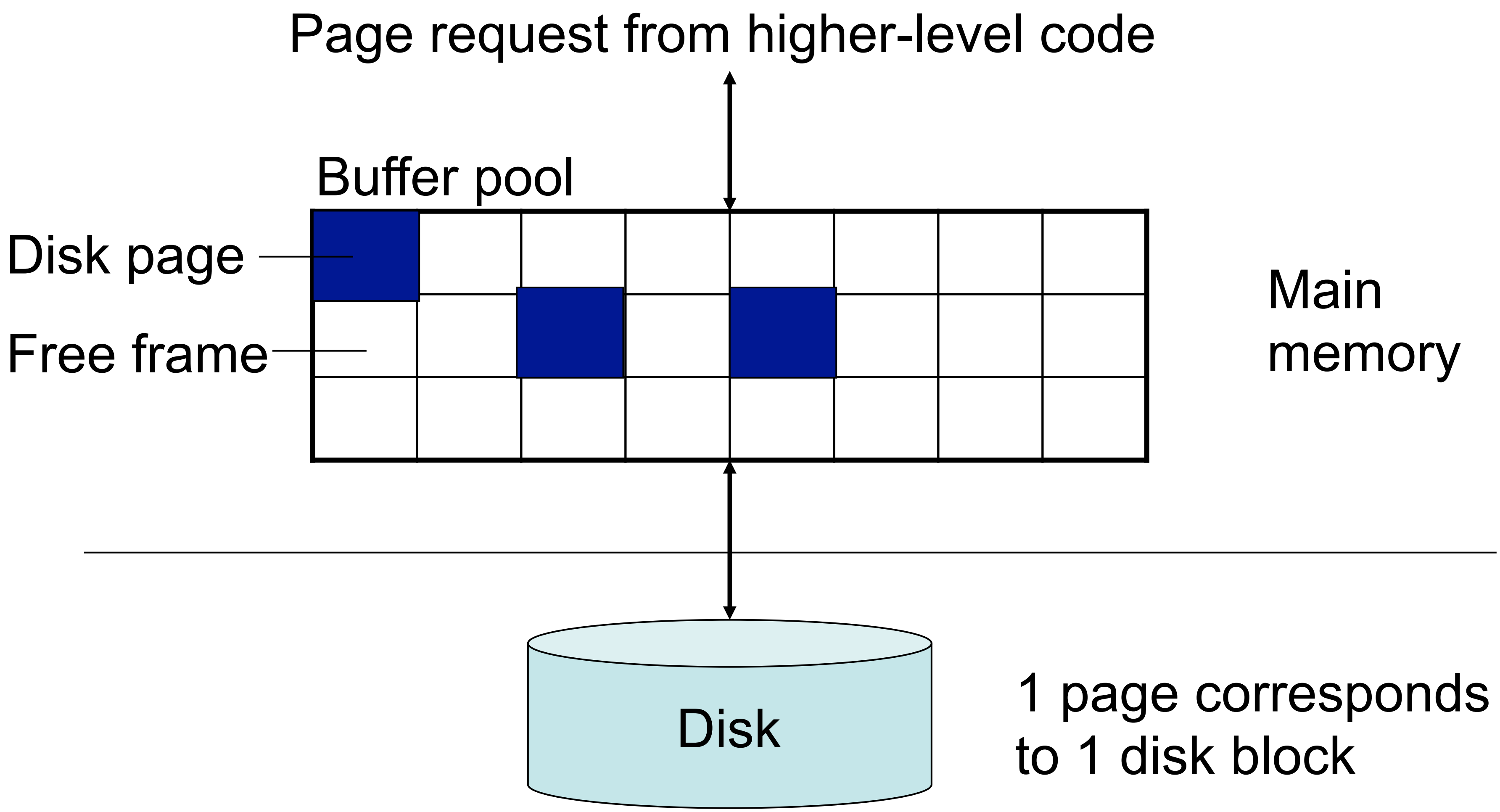
Crash !

What do we do now?

# Handling Failures

- Types of failures
  - Transaction failure
  - System failure
  - Media failure -> we will not talk about this now
- Required capability: undo and redo
- Challenge: buffer manager
  - Changes performed in memory
  - Changes written to disk only from time to time

# Impact of Buffer Manager



# Primitive Operations

- READ(X,t)
  - copy value of data item X to transaction local variable t
- WRITE(X,t)
  - copy transaction local variable t to data item X

- INPUT(X)
  - read page containing data item X to memory buffer
- OUTPUT(X)
  - write page containing data item X to disk

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t);

| | Transaction | Buffer pool | | Disk | |
|---|---|---|---|---|---|
| Action | t | Mem A | Mem B | Disk A | Disk B |
| INPUT(A) | | | | 8 | 8 |
| READ(A,t) | | | | | |
| t:=t*2 | | | | | |
| WRITE(A,t) | | | | | |
| INPUT(B) | | | | | |
| READ(B,t) | | | | | |
| t:=t*2 | | | | | |
| WRITE(B,t) | | | | | |
| OUTPUT(A) | | | | | |
| OUTPUT(B) | | | | | |

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

| | Transaction | Buffer pool | | Disk | |
|---|---|---|---|---|---|
| Action | t | Mem A | Mem B | Disk A | Disk B |
| INPUT(A) | | 8 | | 8 | 8 |
| READ(A,t) | | | | | |
| t:=t*2 | | | | | |
| WRITE(A,t) | | | | | |
| INPUT(B) | | | | | |
| READ(B,t) | | | | | |
| t:=t*2 | | | | | |
| WRITE(B,t) | | | | | |
| OUTPUT(A) | | | | | |
| OUTPUT(B) | | | | | |

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

| | Transaction | Buffer pool | | Disk | |
|---|---|---|---|---|---|
| Action | t | Mem A | Mem B | Disk A | Disk B |
| INPUT(A) | | 8 | | 8 | 8 |
| READ(A,t) | 8 | 8 | | 8 | 8 |
| t:=t*2 | 16 | 8 | | 8 | 8 |
| WRITE(A,t) | | | | | |
| INPUT(B) | | | | | |
| READ(B,t) | | | | | |
| t:=t*2 | | | | | |
| WRITE(B,t) | | | | | |
| OUTPUT(A) | | | | | |
| OUTPUT(B) | | | | | |

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t);

| | Transaction | Buffer pool | | Disk | |
|---|---|---|---|---|---|
| Action | t | Mem A | Mem B | Disk A | Disk B |
| 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 |
| INPUT(B) | | | | | |
| READ(B,t) | | | | | |
| t:=t*2 | | | | | |
| WRITE(B,t) | | | | | |
| OUTPUT(A) | | | | | |
| OUTPUT(B) | | | | | |

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t);

| | Transaction | Buffer pool | | Disk | |
|---|---|---|---|---|---|
| Action | t | Mem A | Mem B | Disk A | Disk B |
| 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 |
| INPUT(B) | 16 | 16 | 8 | 8 | 8 |
| READ(B,t) | | | | | |
| t:=t*2 | | | | | |
| WRITE(B,t) | | | | | |
| OUTPUT(A) | | | | | |
| OUTPUT(B) | | | | | |

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

| Action | t | Mem A | Mem B | Disk A | Disk B |
|---|---|---|---|---|---|
| | Transaction | Buffer pool | | Disk | |
| 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 |
| 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) | | | | | |
| OUTPUT(A) | | | | | |
| OUTPUT(B) | | | | | |

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t);

| | Transaction | Buffer pool | | Disk | |
|---|---|---|---|---|---|
| Action | t | Mem A | Mem B | Disk A | Disk B |
| 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 |
| 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 |
| OUTPUT(A) | | | | | |
| OUTPUT(B) | | | | | |

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t);

| | Transaction | Buffer pool | | Disk | |
|---|---|---|---|---|---|
| Action | t | Mem A | Mem B | Disk A | Disk B |
| 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 |
| 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 |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | | | | | |

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t);

| | Transaction | Buffer pool | | Disk | |
|---|---|---|---|---|---|
| Action | t | Mem A | Mem B | Disk A | Disk B |
| 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 |
| 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 |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 |

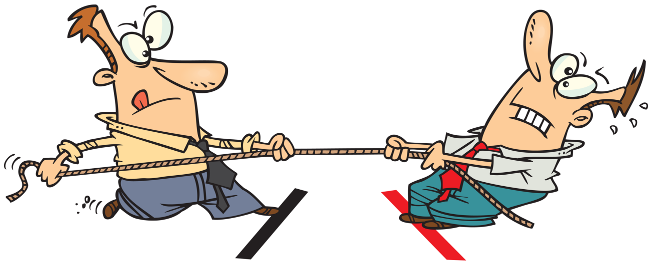
# Buffer Manager Policies

- **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?

- Easiest for recovery: NO-STEAL/FORCE

- Highest performance: STEAL/NO-FORCE



# Outline

- **Review of ACID properties**
  - Today we will cover techniques for ensuring atomicity and durability in face of failures
- **Review of buffer manager and its policies**
- **Write-ahead log**
- **ARIES method for failure recovery**

# Solution: Use a Log

- **Log: append-only file containing log records**
- Enables the use of STEAL and NO-FORCE
- For every update, commit, or abort operation
  - Write a log record
  - Multiple transactions run concurrently, log records are interleaved
- After a system crash, use log to:
  - Redo transactions that did commit
  - Undo other transactions that didn’t commit

# Write-Ahead Log

- All log records pertaining to a page are written to disk before the page is overwritten on disk
- All log records for transaction are written to disk before the transaction is considered committed
  - Why is this faster than FORCE policy?
- **Committed transaction**: transactions whose commit log record has been written to disk

# 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

# Granularity in ARIES

- *Physiological logging*
  - Log records refer to a single page
  - But record logical operation within the page

- Page-oriented logging for REDO
  - Necessary since can crash in middle of complex operation
- Logical logging for UNDO
  - Enables **tuple-level locking**!
  - Must do logical undo because ARIES will only undo loser transactions (this also facilitates ROLLBACKs)

# ARIES Method

**Recovery from a system crash is done in 3 passes:**

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

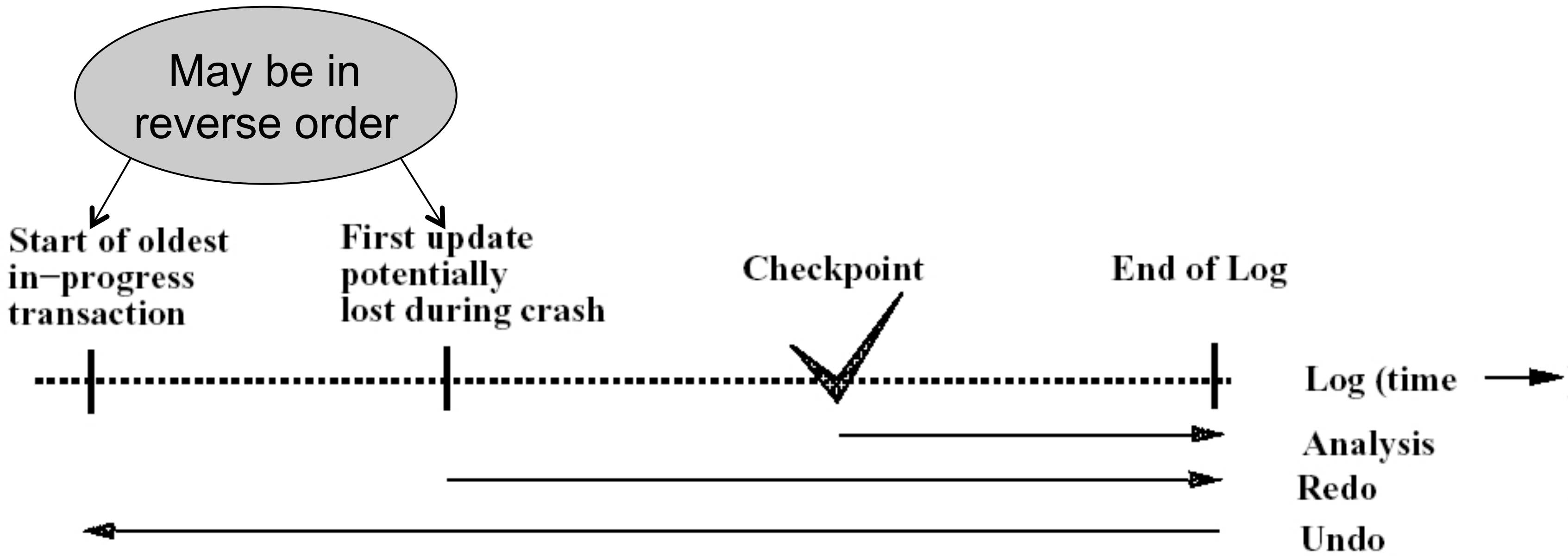



Figure 3: The Three Passes of ARIES Restart

[Franklin97]

# ARIES Method Elements

- Each page contains a **pageLSN**
  - Log Sequence Number of log record for latest update to that page
  - Will serve to determine if an update needs to be redone
- Physiological logging
  - page-oriented REDO
    - Possible because will always redo all operations in order
  - logical UNDO
    - Needed because will only undo some operations

# ARIES Method Data Structures

- Active transactions table
  - Lists all running transactions (active transactions)
  - With lastLSN, most recent update by transaction
- Dirty page table
  - Lists all dirty pages
  - With recoveryLSN, first LSN that caused page to become dirty
- Write ahead log contains log records
  - LSN, prevLSN: previous LSN for same transaction
  - other attributes

# ARIES Data Structures

## Dirty pages

| pageID | recLSN |
|---|---|
| P5 | 102 |
| P6 | 103 |
| P7 | 101 |

## Log

| 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

| | | |
|---|---|---|
| | | |
| | | |
| P5<br>PageLSN=104 | P6<br>PageLSN=103 | P7<br>PageLSN=101 |
| | | |

# The LSN

- Each log entry receives a unique *Log Sequence Number*, LSN
  - The LSN is written in the log entry
  - Entries belonging to the same transaction are chained in the log via **prevLSN**
  - LSN's help us find the end of a circular log file:

After crash, log file = (22, 23, 24, 25, 26, 18, 19, 20, 21)
Where is the end of the log ?

# ARIES Method Details

- **Let's walk through example on board**
  - Please take notes
- Steps under normal operations
  - Add log record
  - Update transactions table
  - Update dirty page table
  - Update pageLSN

# Checkpoints

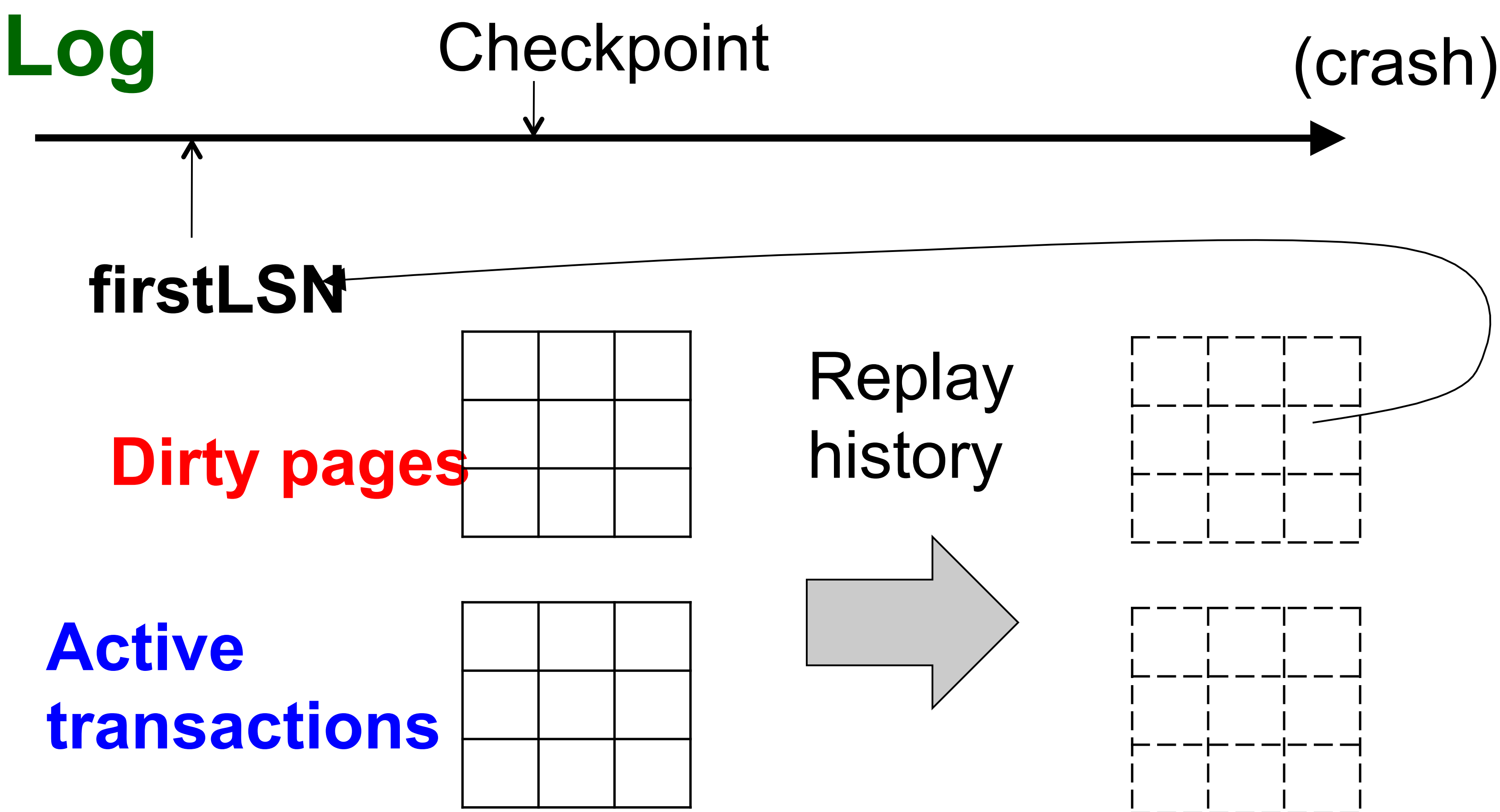
- Write into the log
  – Contents of transactions table
  – Contents of dirty page table
- Enables REDO phase to restart from earliest recoveryLSN in dirty page table
  – Shortens REDO phase
- But, effectiveness is limited by dirty pages
  – Sol: Background process periodically sends dirty pages to disk

# 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 beginning (or checkpoint)
    - Only update the two data structures
  - Compute: **firstLSN** = smallest of all recoveryLSN

# 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

- If affected page is not in Dirty Page Table then **do not update**
- If recoveryLSN > LSN, then **no update**
- Read page from disk;
  If **pageLSN** >= LSN, then **no update**
- Otherwise perform update

# 3. Undo Phase

Main principle: “logical” undo

- Start from the end of the 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
- While **ToUndo** not empty:
  - Choose most recent (largest) LSN in **ToUndo**
  - If LSN = regular record: undo; write a CLR where CLR.undoNextLSN = LSN.prevLSN; if LSN.prevLSN not null, insert in **ToUndo** otherwise, write <END TRANSACTION> in log
  - If LSN = CLR record: (don't undo !)
    if CLR.**undoNextLSN** not null, insert in **ToUndo**
    otherwise, write <END TRANSACTION> in log

# Handling Crashes during Undo

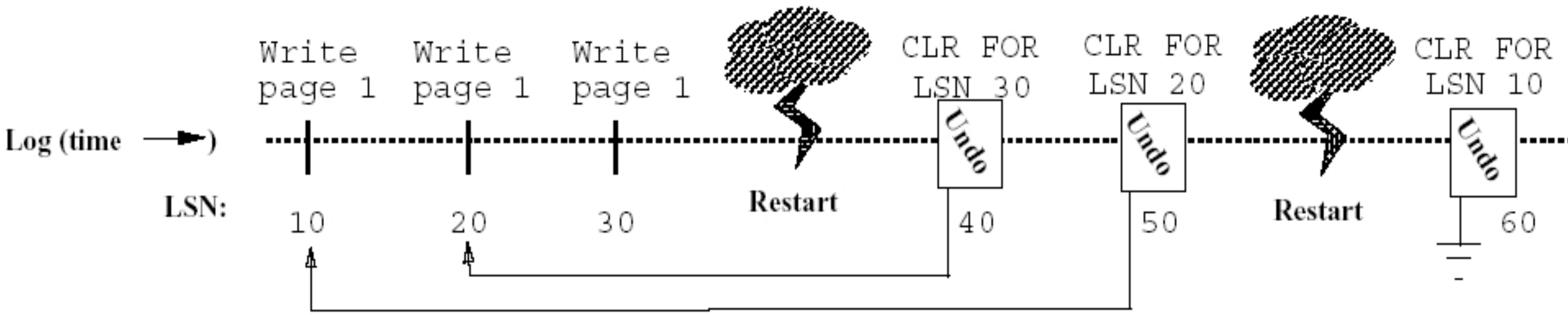



Figure 4: The Use of CLRs for UNDO

[Figure 4 from Franklin97]

# Summary

- Transactions are a useful abstraction
- They simplify application development
- DBMS must maintain ACID properties in face of
  - Concurrency
  - Failures