Big Data Infrastructures

Philippe Cudré-Mauroux

Fall 2018

Lecture 8 – Transactions, recovery

Outline

- **Review of ACID properties**
  - Today we will cover techniques for ensuring atomicity and durability in face of failures
- **Concurrent operations**
- **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 (only valid tuples are written)
- 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
  - Isolation property

- Failures can occur at any time
  - Atomicity and durability properties

Different Types of Problems

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

          DELETE Product
          WHERE price <=0.99

Client 2: SELECT count(*)
          FROM Product

          SELECT count(*)
          FROM SmallProduct
```

What could go wrong ?

Inconsistent reads

Different Types of Problems

Client 1: UPDATE SET Account.amount = 1000000000
WHERE Account.number = ‘my-account’

Aborted by system

Client 2: SELECT Account.amount
FROM Account
WHERE Account.number = ‘my-account’

What could go wrong ? Dirty reads

Different Types of Problems

Client 1:

```
UPDATE Product
SET Price = Price – 1.99
WHERE pname = ‘Gizmo’
```

Client 2:

```
UPDATE Product
SET Price = Price*0.5
WHERE pname=‘Gizmo’
```

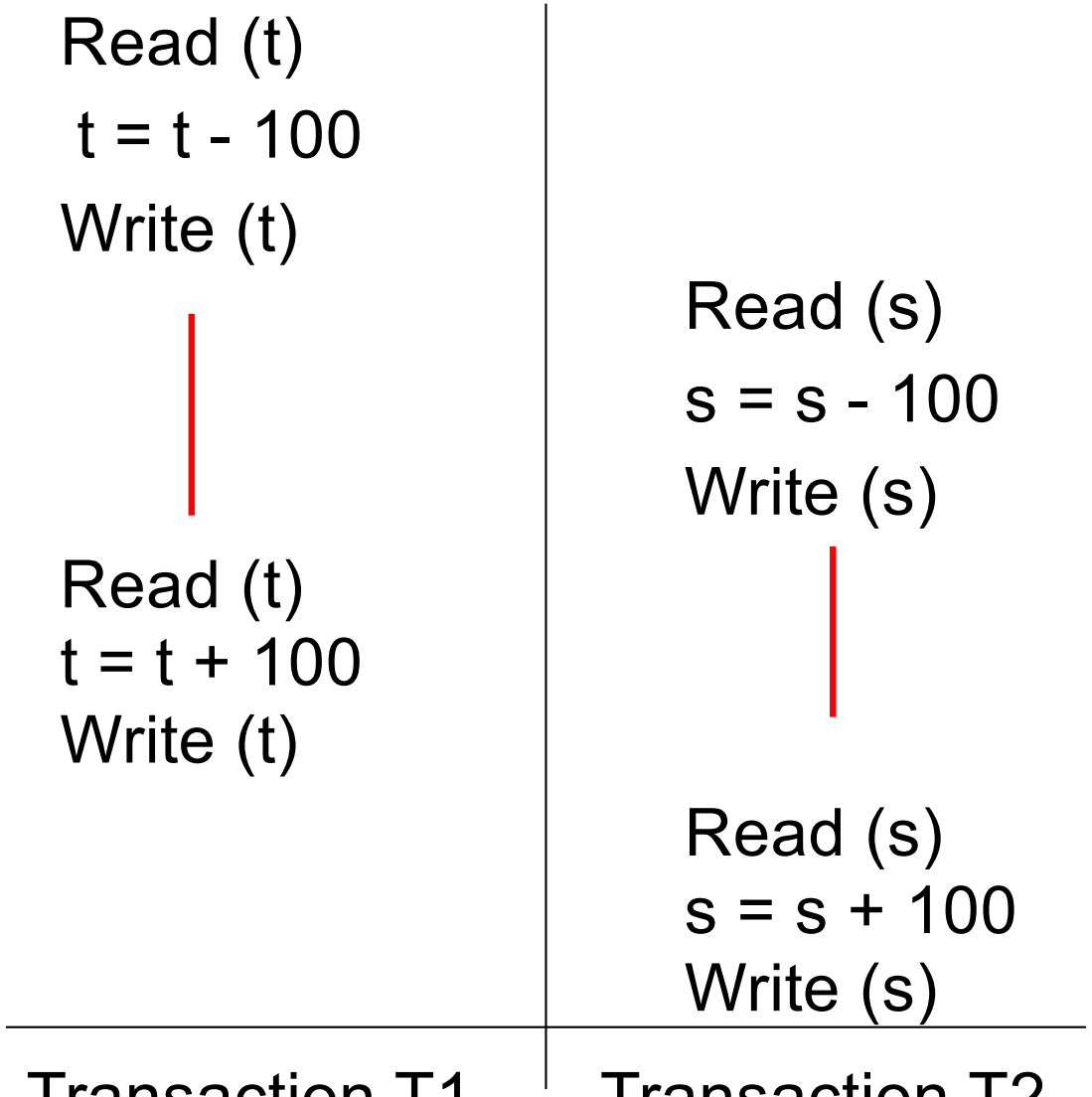
What could go wrong ? Lost update

Serializable Execution

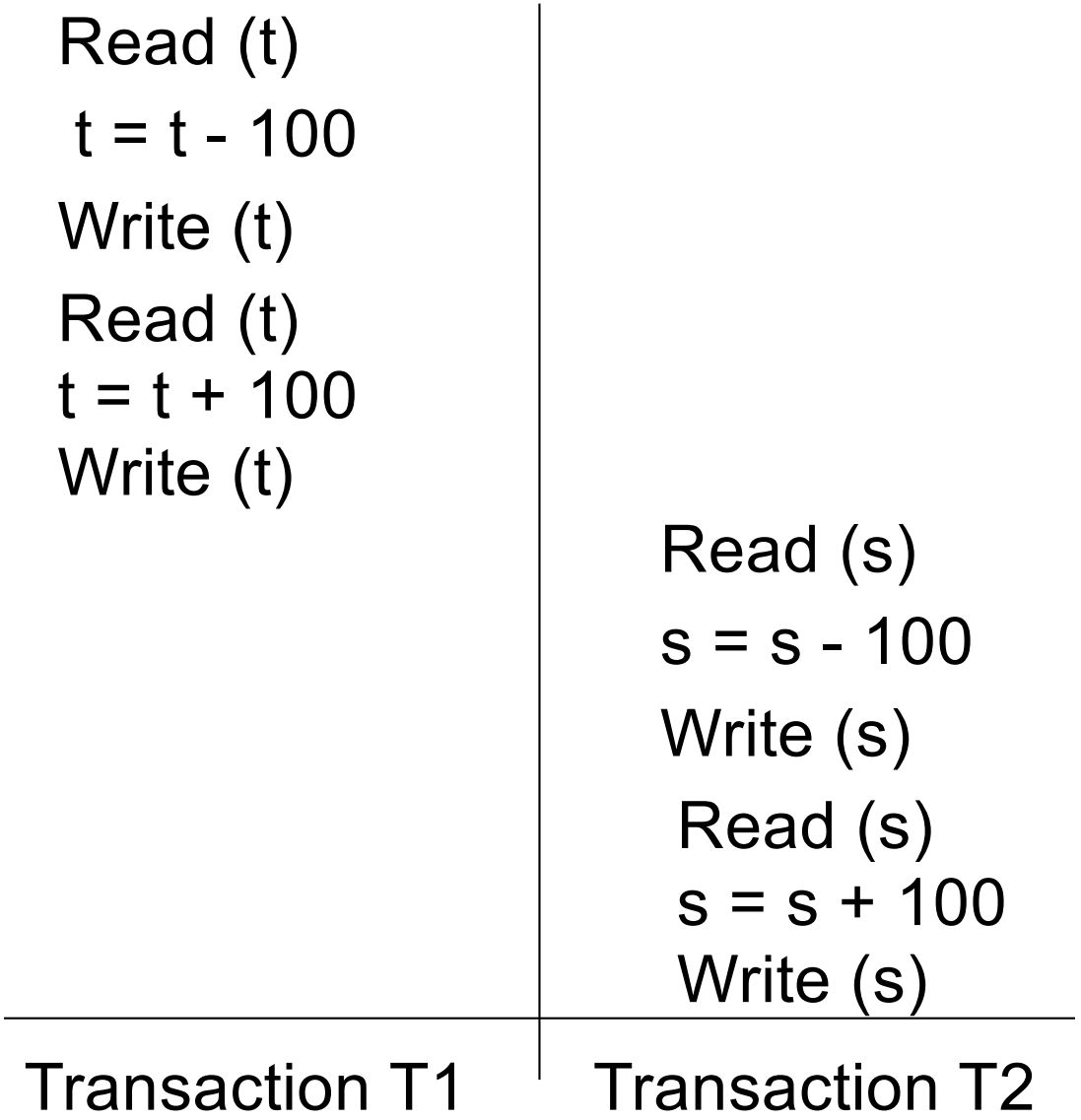
- **Serializability**: ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order

- A schedule is called "correct"" if we can find a serial schedule that is "equivalent" to it

Is This Serializable?



Equivalent Serial Schedule



Implementation: Locking

- A lock is a synchronization mechanism to limit access to a resource
- Can serve to enforce serializability
- Lock contention
  - occurs whenever one process or thread attempts to acquire a lock held by another process or thread
- Two types of locks: **Shared (read-only) and Exclusive (read and write)**

Degrees of Isolation

- Isolation level “serializable” (i.e. ACID)
  - Golden standard
  - Requires strict locking (*2PL*)
  - But often too inefficient
  - Imagine there are only a few update operations and many long read operations

- Weaker isolation levels
  - Sacrifice correctness for efficiency
  - Often used in practice (often **default**)
  - Sometimes are hard to understand

What Could Go Wrong?

- Concurrent operations
  - Isolation property
- Failures can occur at any time
  - Atomicity 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
```

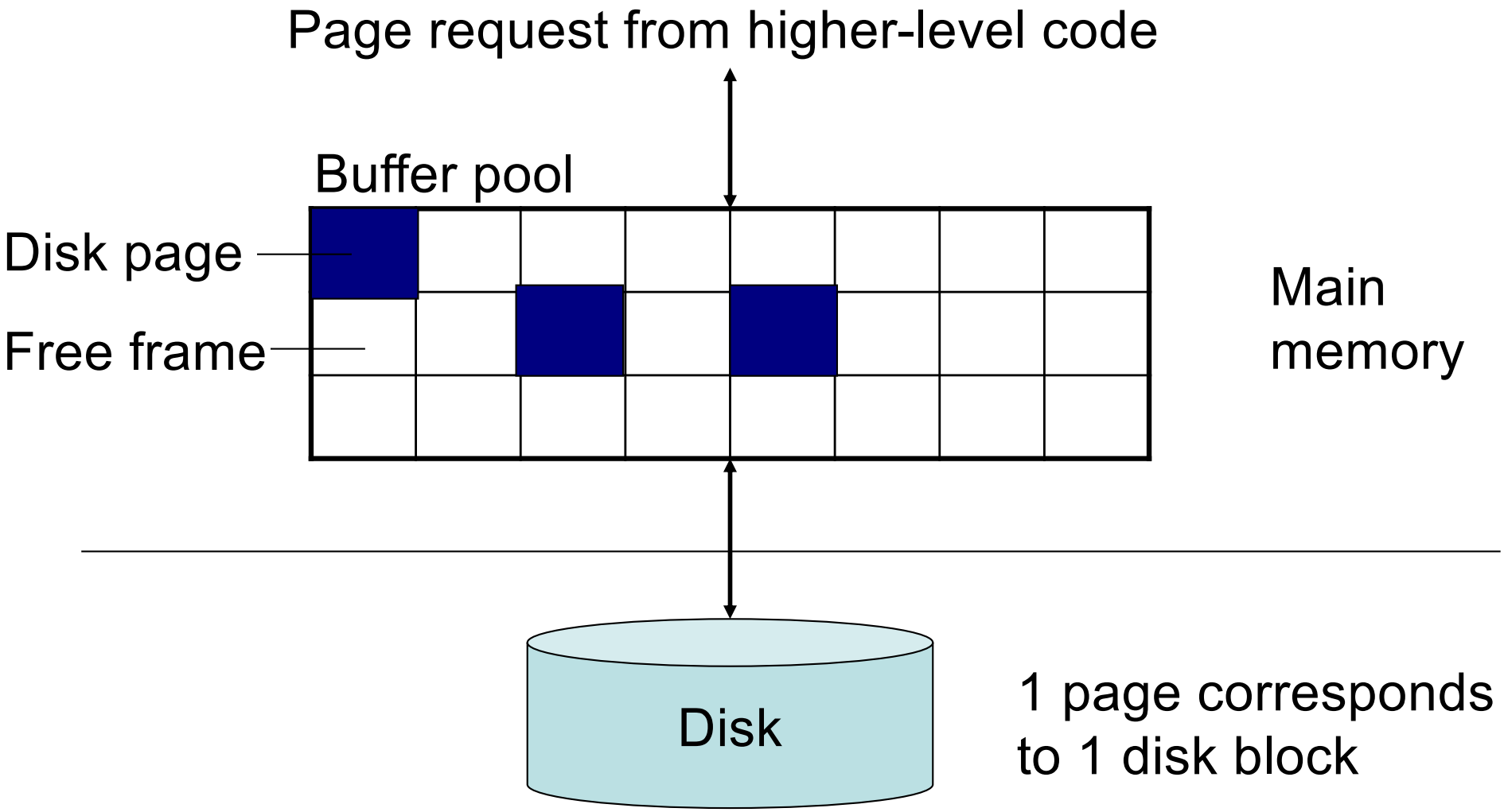
Crash !

What do we do now?

Handling Failures

- Types of failures
  - Transaction failure
  - System failure
  - Media failure -> we will not talk about this
- Required capability: undo and redo
- Challenge: buffer manager (and memory hierarchy)
  - 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);
```

	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)					
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**
  - The buffer manager allows an update made by an uncommitted transaction to overwrite the most recent committed value of a data item on non-volatile storage
  - The opposite is NO-STEAL
- **FORCE**
  - The buffer manager ensures that all updates made by a transaction are reflected on non-volatile storage before the transaction is allowed to commit
  - The opposite is NO-FORCE
- 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 physical, logical, or physiological log record (cf. later)
  - Note: multiple transactions run concurrently, log records are interleaved
- After a system crash, use log to:
  - Redo some transaction 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 a 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

ARIES Method

- Write-Ahead Log

- Three pass algorithm
  - **Analysis pass**
    - Figure out what was going on at time of crash
    - List of dirty pages and active transactions
  - **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
  - **Undo pass**
    - Remove effects of all uncommitted transactions
    - Log changes during undo in case of another crash during undo

ARIES Method Illustration

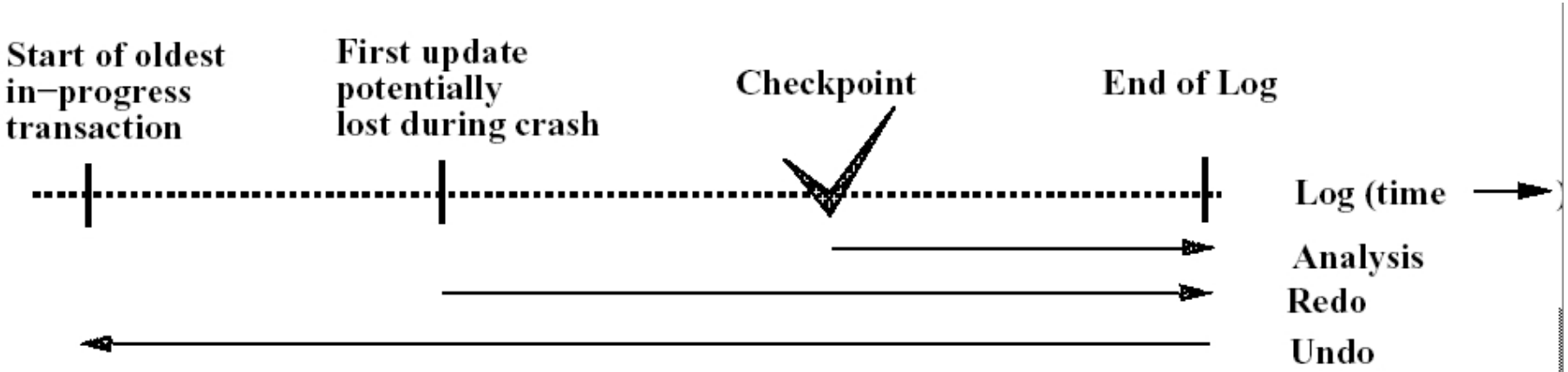


Figure 3: The Three Passes of ARIES Restart

[Figure 3 from 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*

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

ARIES Method Details

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

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 transactions table and dirty pages table
  - Reprocess the log from the beginning (or checkpoint)
    - Only update the two data structures
  - Find oldest recoveryLSN (firstLSN) in dirty pages tables

Redo Phase

- Goal: redo all updates since firstLSN

- For each log record
  - If affected page is not in Dirty Page Table then **do not update**
  - If affected page is in Dirty Page Table but recoveryLSN > LSN of record, then **no update**
  - Else if pageLSN > LSN, then **no update**
    - Note: only condition that requires reading page from disk
  - Otherwise perform update

Undo Phase

- Goal: undo effects of aborted transactions
- Identifies all loser transactions in trans. table
- Scan log backwards
  - Undo all operations of loser transactions
  - Undo each operation unconditionally
  - All ops. logged with compensation log records (CLR)
  - Never undo a CLR
    - Look-up the UndoNextLSN and continue from there

Handling Crashes during Undo

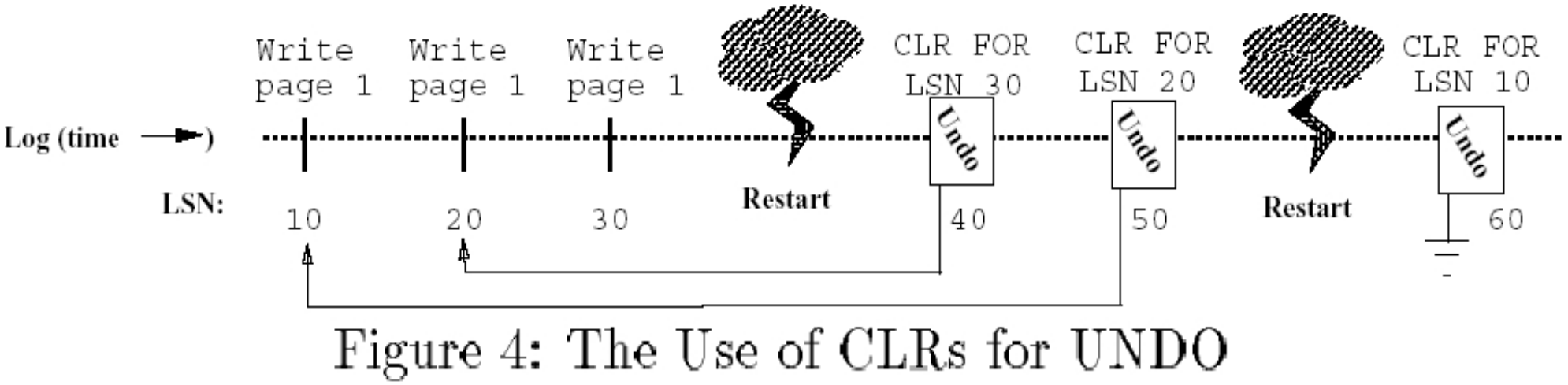


Figure 4: The Use of CLRs for UNDO

[Figure 4 from Franklin97]

Example

- After a crash, we find the following log:

```
0   BEGIN CHECKPOINT
5   END CHECKPOINT (EMPTY XACT TABLE AND DPT)
10  T1: UPDATE P1 (OLD: YYY NEW: ZZZ)
15  T1: UPDATE P2 (OLD: WWW NEW: XXX)
20  T2: UPDATE P3 (OLD: UUU NEW: VVV)
25  T1: COMMIT
30  T2: UPDATE P1 (OLD: ZZZ NEW: TTT)
```

Analysis Phase

- Scan forward through the log starting at LSN 0.
- LSN 5: Initialize XACT table and DPT to empty.
- LSN 10: Add (T1, LSN 10) to XACT table. Add (P1, LSN 10) to DPT.
- LSN 15: Set LastLSN=15 for T1 in XACT table. Add (P2, LSN 15) to DPT.
- LSN 20: Add (T2, LSN 20) to XACT table. Add (P3, LSN 20) to DPT.
- LSN 25: Change T1 status to "Commit" in XACT table
- LSN 30: Set LastLSN=30 for T2 in XACT table. Add (P1, LSN 30) to DPT.

Redo Phase

- Scan forward through the log starting at LSN 10.
- LSN 10: Read page P1, check PageLSN stored in the page. If PageLSN<10, redo LSN 10 (set value to ZZZ) and set the page's PageLSN=10.
- LSN 15: Read page P2, check PageLSN stored in the page. If PageLSN<15, redo LSN 15 (set value to XXX) and set the page's PageLSN=15.
- LSN 20: Read page P3, check PageLSN stored in the page. If PageLSN<20, redo LSN 20 (set value to VVV) and set the page's PageLSN=20.
- LSN 30: Read page P1 if it has been flushed, check PageLSN stored in the page. If PageLSN<30, redo LSN 30. Redo LSN 30 (set value to TTT) and set the page's PageLSN=30.

Undo Phase

- T2 must be undone. Put LSN 30 in ToUndo.
- Write Abort record to log for T2
- LSN 30: Undo LSN 30 - write a CLR for P1 with "set P1=ZZZ" and undonextLSN=20. Write ZZZ into P1. Put LSN 20 in ToUndo.
- LSN 20: Undo LSN 20 - write a CLR for P3 with "set P3=UUU" and undonextLSN=NULL. Write UUU into P3.

Summary

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