CSE 544
Principles of Database Management Systems

Magdalena Balazinska

Fall 2007

Lecture 11 - Transactions: recovery

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 (isolation property)
- Failures can occur at any time
  – Today (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
```

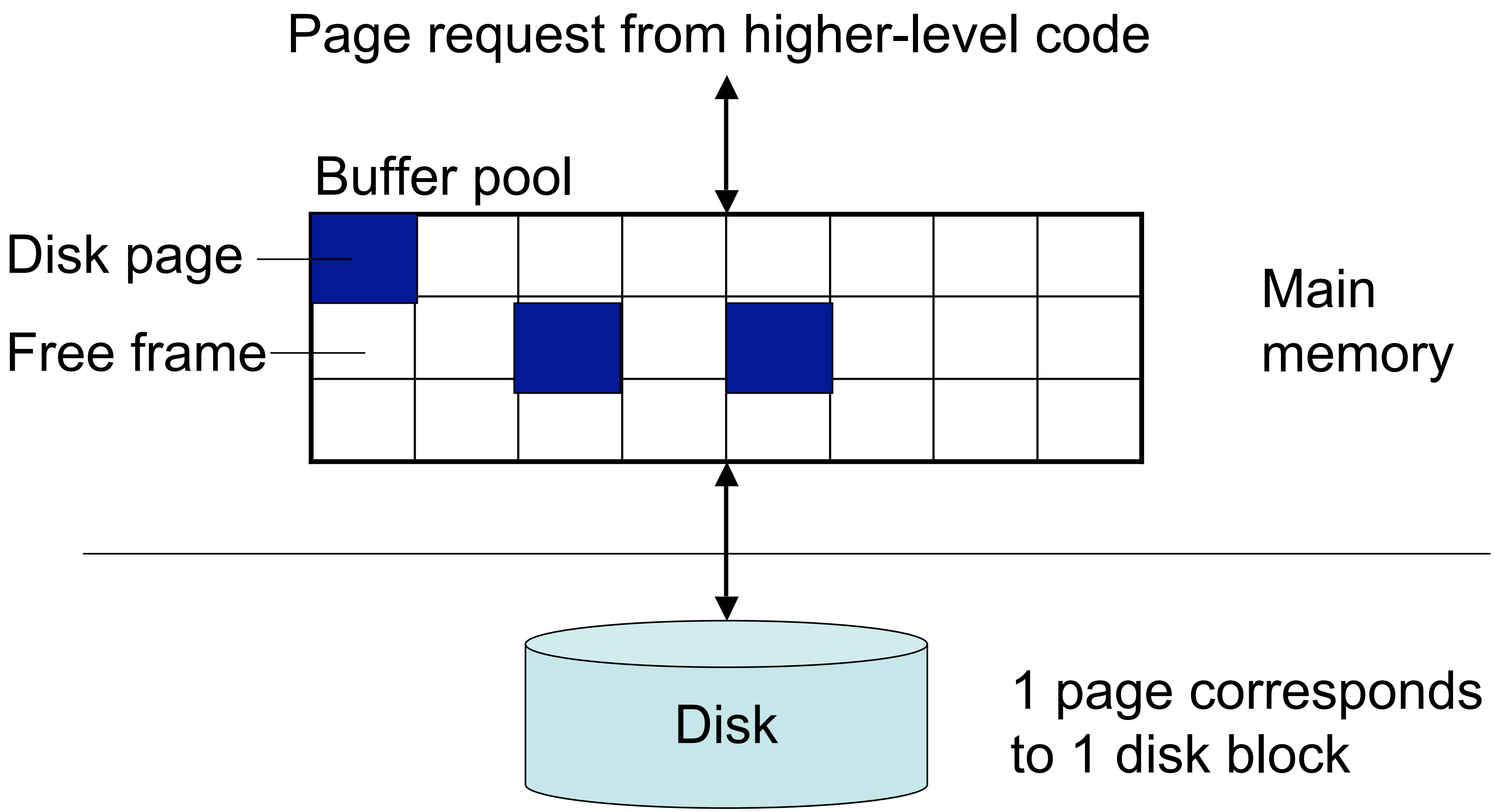
Crash !

What do we do now?

Handling Failures

- Types of failures
  - Transaction failure
  - System failure
  - Media failure -> we will not talk about this 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);
```

	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 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 physical, logical, or physiological log record
  - 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 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

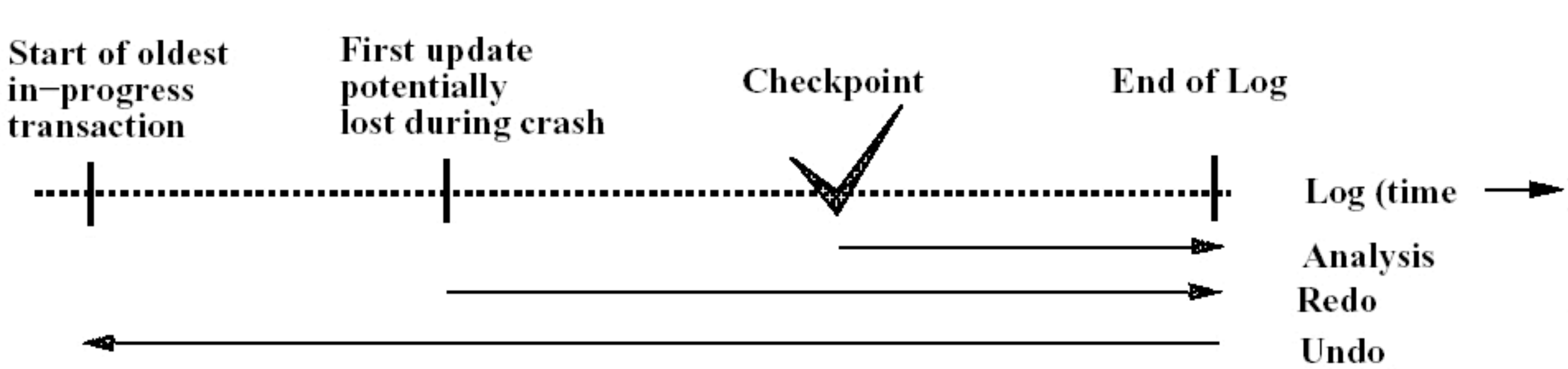


Figure 3: The Three Passes of ARIES Restart

[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
- Dirty page table
  - Lists all dirty pages
  - With recoveryLSN, LSN that caused page to be dirty
- Write ahead log contains log records
  - LSN, prevLSN: previous LSN for same transaction
  - other attributes

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

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

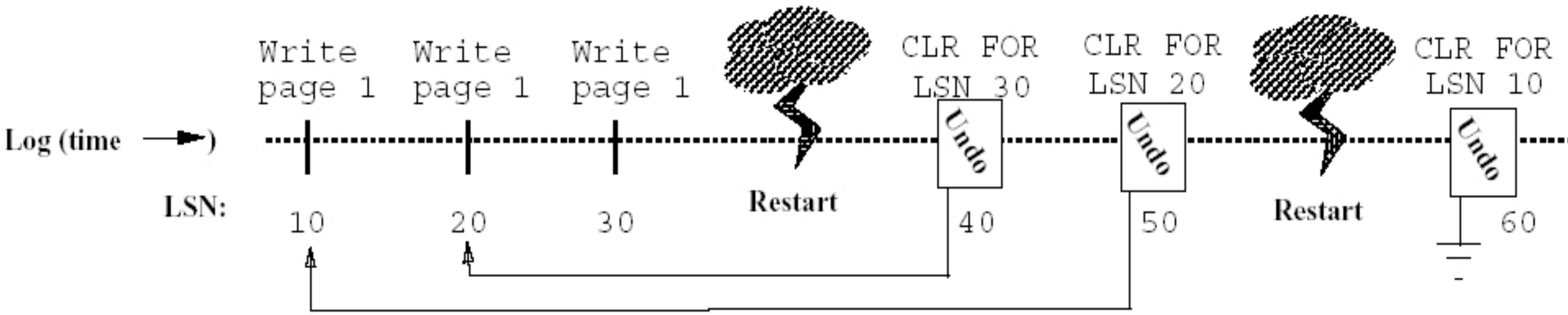


Figure 4: The Use of CLRs for UNDO

[Figure 4 from Franklin97]

Summary

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