Intro to DB

CHAPTER 16
RECOVERY SYSTEM

Chapter 16: Recovery System

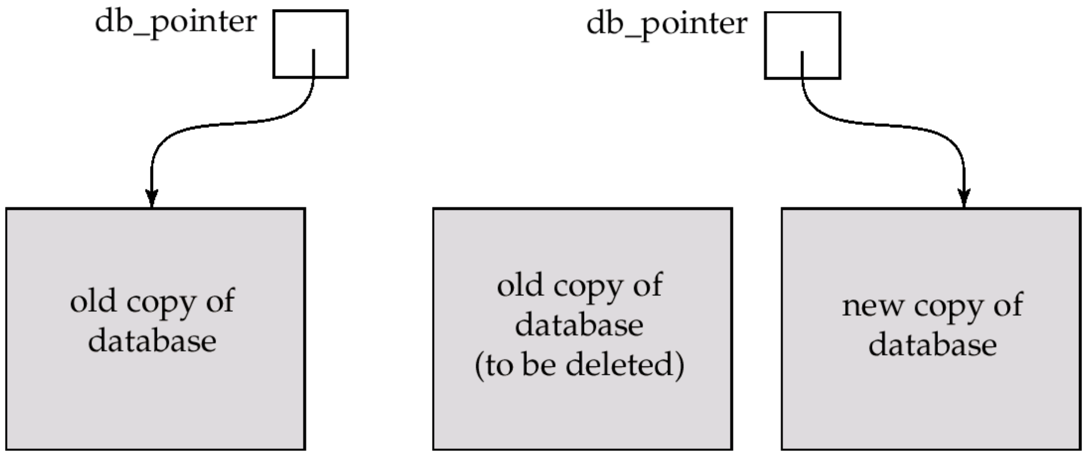
- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Recovery Algorithm
- Buffer Management
- Failure with Loss of Nonvolatile Storage
- Early Lock Release and Logical Undo Operations
- Advanced Recovery Techniques
- Remote Backup Systems

© Silberschatz, Korth and Sudarshan

Copyright © by S.-g. Lee

Implementation of Atomicity and Durability

- The   component of a DB system implements the support for atomicity and durability.
- The *shadow-database* scheme:
  - assume that only one transaction is active at a time.
  - a pointer called **db_pointer** always points to the current consistent copy of the database.
  - all updates are made on a *shadow copy* of the database



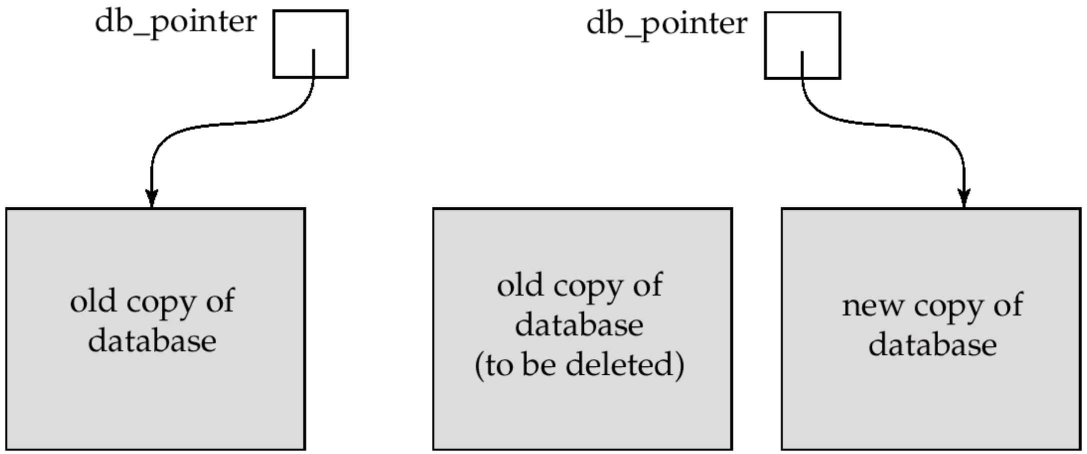
(a) Before update

(b) After update

The Shadow Database Scheme

- **db_pointer** is made to point to the updated shadow copy only after
  - the transaction reaches partial commit and
  - all updated pages have been flushed to disk.
- in case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.

- Assumes disks will not fail
- extremely inefficient for large databases
  - executing a single transaction requires copying the *entire* database
  - Useful for text editors
- Better schemes exist



(a) Before update (b) After update

Failure Classification

- Transaction failure
  - *Logical errors*: transaction cannot complete due to some internal error condition
  - *System errors*: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- System crash
  - a power failure or other hardware or software failure causes the system to crash.
- Disk failure
  - a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures
- non-volatile storage contents are assumed to not have been corrupted by system crash
  - Database systems have numerous integrity checks to prevent corruption of disk data

Storage Structure

- Volatile storage:
  - does not survive system crashes
  - examples: main memory, cache memory
- Nonvolatile storage:
  - survives system crashes
  - examples: disk, tape, flash memory,
    non-volatile (battery backed up) RAM
- **Stable storage**:
  - a theoretical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media

Data Access

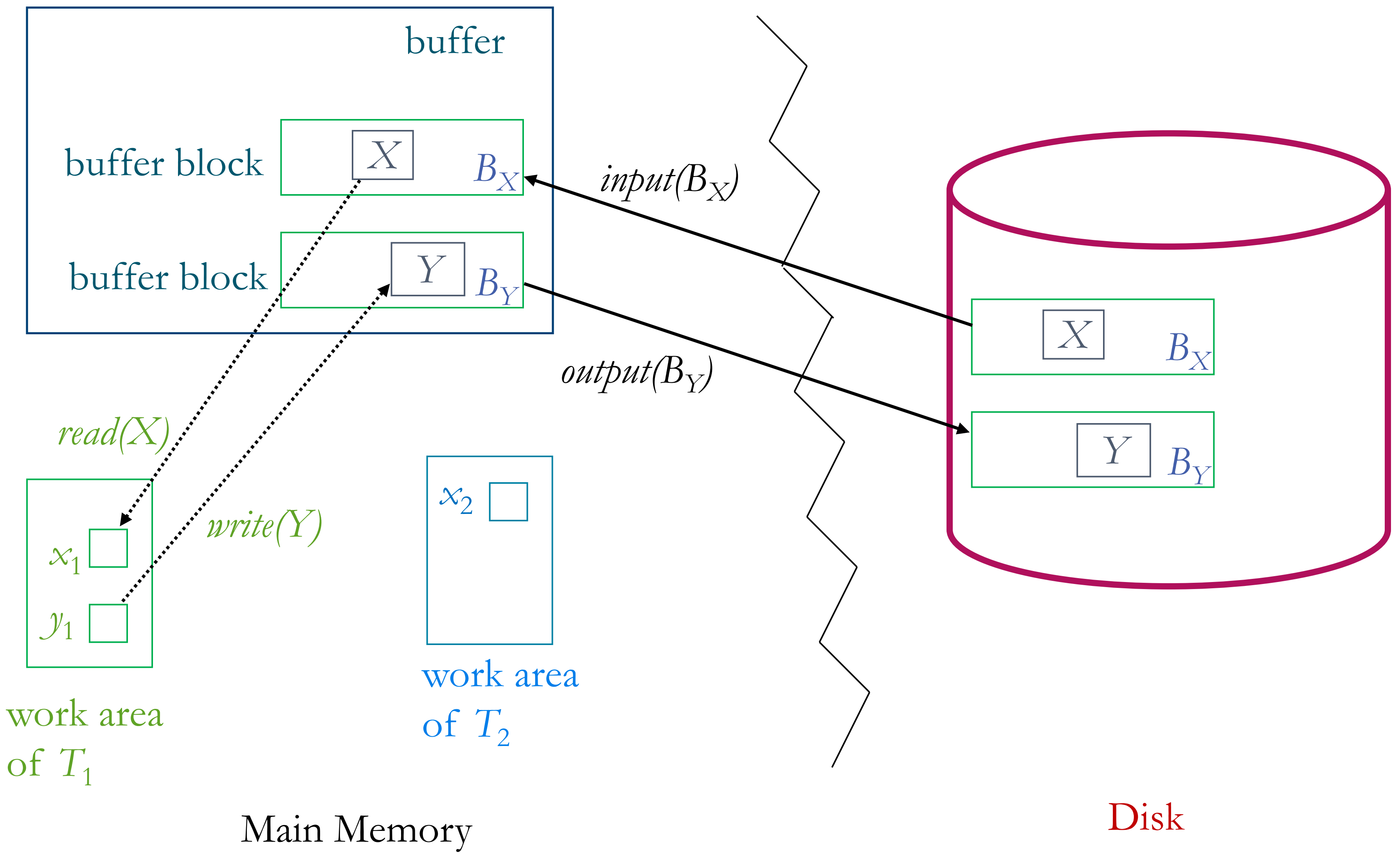
- Data blocks
  - Physical blocks: blocks residing on the disk
  - Buffer blocks: blocks residing temporarily in main memory (disk buffer).
- Each transaction T_i has its private work-area
  - local copies of all data items accessed and updated by it are kept here
  - T_i's local copy of a data item X is called x_i.
- Block movements between disk and main memory:
  - **input**(B) transfers the physical block B to main memory.
  - **output**(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.
- We assume that no data item spans two or more blocks.

Data Access (Cont.)

- Transaction transfers data items between system buffer blocks and its private work-area using the following operations :
  - **read**(X)
    - If B_X in which X resides is not in memory, issue **input**(B_X)
    - assign to the local variable x_i the value of X from the buffer block
  - **write**(X)
    - If B_X in which X resides is not in memory, issue **input**(B_X)
    - assign the value x_i to X in the buffer block.
- Transactions
  - perform **read**(X) when accessing X for the first time;
  - All subsequent accesses are to the local copy x_i.
  - After last access, transaction executes **write**(X) if updated.
- output(B_X) need not immediately follow write(X)
  - System can perform the **output** operation when it deems fit.

Original Slides:
© Silberschatz, Korth and Sudarshan

Example of Data Access



Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
- Recovery algorithms have two parts
  1. Actions taken $\boxed{}$ processing to ensure enough information exists to recover from failures
  2. Actions taken $\boxed{}$ to recover the database contents to a state that ensures atomicity, consistency and durability

Log-Based Recovery

- **Log**
  - a sequence of that describe update activities on the database
  - Log is kept on stable storage
- Log records
  - When transaction T_i starts: <T_i **start**>
  - *Before* T_i executes write(X): <T_i, X, V_1, V_2>

    V_1: the value of X before the write

    V_2: the value to be written to X.
  - When T_i finishes its last statement: <T_i **commit**>
  - When T_i rolls back and finishes its roll back process: <T_i **abort**>
- Assume that
  - transactions execute serially
  - log records are written directly to stable storage (they are not buffered)

Immediate Database Modification

- Allows database updates of an uncommitted transaction to be made as the writes are issued
- Logging for Immediate DB Modification
  1. Transaction start: <$\boldsymbol{T_i}$ *start*>
  2. A **write**(X) operation results in
     a. <T_i, X, V_1, V_2> being written to log (undoing may be needed)
     b. followed by the write operation
  3. When T_i partially commits, <T_i **commit**> is written to the log
- Output of updated blocks can take place $\boxed{}$ transaction commit
  - Order in which blocks are output can be different from the order in which they are written

Immediate Database Modification (Cont.)

Log	Write	Output
<T_0 start>		
<T_0, A, 1000, 950>		
	A = 950	
<T_o, B, 2000, 2050>		
	B = 2050	
<T_0 commit>		
<T_1 start>		
<T_1, C, 700, 600>		
	C = 600	
		B_B, B_C
<T_1 commit>		
		B_A

(B_X denotes block containing X)

Original Slides:
© Silberschatz, Korth and Sudarshan

Copyright © by S.-g. Lee

Undo & Redo Operations

- **undo**(T_i): restores the value of all data items updated by T_i to their old values
  - going backwards from the last log record for T_i
- **redo**(T_i): sets the value of all data items updated by T_i to the new values,
  - going forward from the first log record for T_i
- Both operations must be *idempotent*
  - even if the operation is executed multiple times, the effect is the same as if it is executed once
  - needed since operations may get re-executed during recovery

Recovery Logic

Example

- The log as it appears at three instances of time.

$<T_0 \text{ start}>$	$<T_0 \text{ start}>$	$<T_0 \text{ start}>$
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0 \text{ commit}>$	$<T_0 \text{ commit}>$
	$<T_1 \text{ start}>$	$<T_1 \text{ start}>$
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1 \text{ commit}>$
(a)	(b)	(c)

- If log on stable storage at time of crash is as in case:

(a)

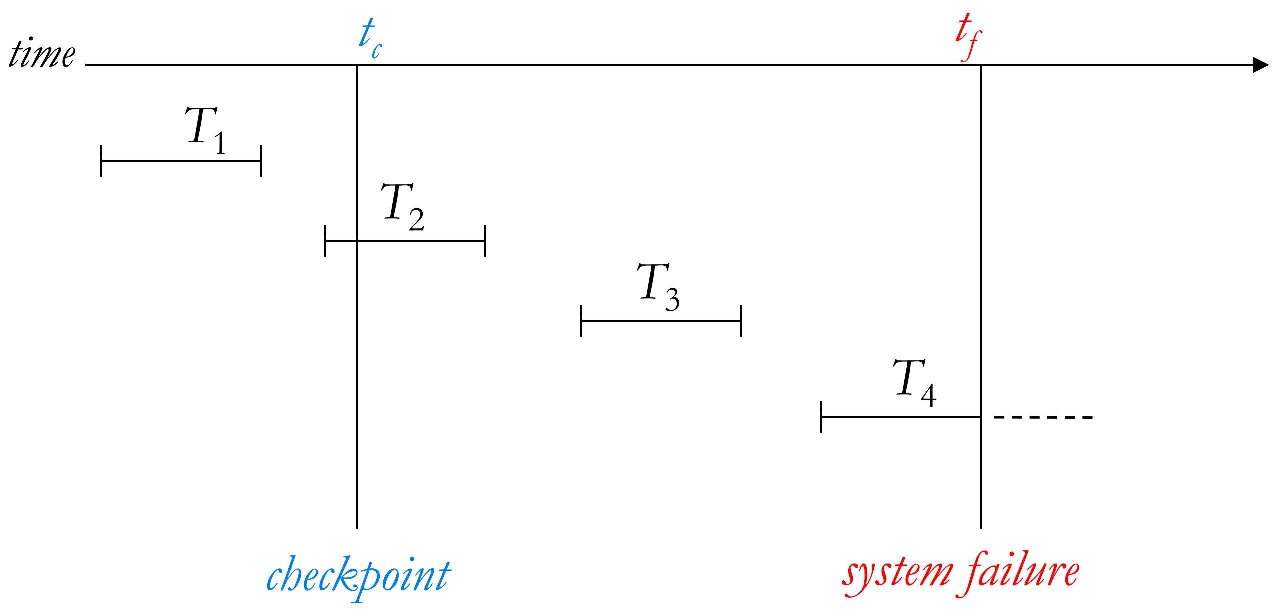
(b)

(c)

Checkpoints

- Problems in previous recovery procedures
  - searching the entire log is time-consuming
  - we might unnecessarily redo transactions which have already output their updates to the database
- Checkpoints reduce
- Checkpoint process
  1. Output all log records currently residing in main memory onto stable storage
  2. Output all modified buffer blocks to the disk
  3. Write a log record < **checkpoint**> onto stable storage

Example of Checkpoints



- (updates already output to disk due to checkpoint)
-
-

Original Slides:
© Silberschatz, Korth and Sudarshan

Copyright © by S.-g. Lee

Log Record Buffering

- Log records are buffered in main memory
  - instead of being output directly to stable storage
  - several log records can be output using a single output operation
- Log records are output to stable storage when
  - a block of log records in the buffer is full, or
  - A *log force* operation is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.

Write-Ahead Logging (WAL)

Rules that must be followed

1. Log records are output to stable storage in the order in which they are created.
2. Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage
3. all log records pertaining to data in that block must have been output to stable storage.

=> called the ***write-ahead logging*** or ***WAL*** rule

END OF CHAPTER 16