



Chapter 19: Recovery System

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



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.
 - **Fail-stop assumption:** non-volatile storage contents are assumed to not be corrupted by system crash
 - Database systems have numerous integrity checks to prevent corruption of disk data
- **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



Recovery Algorithms

- Suppose transaction T_i transfers \$50 from account A to account B
 - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database.
 - A failure may occur after one of these modifications have been made but before both of them are made.
 - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
 - Not modifying the database may result in lost updates if failure occurs just after transaction commits
- Recovery algorithms have two parts
 1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
 2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

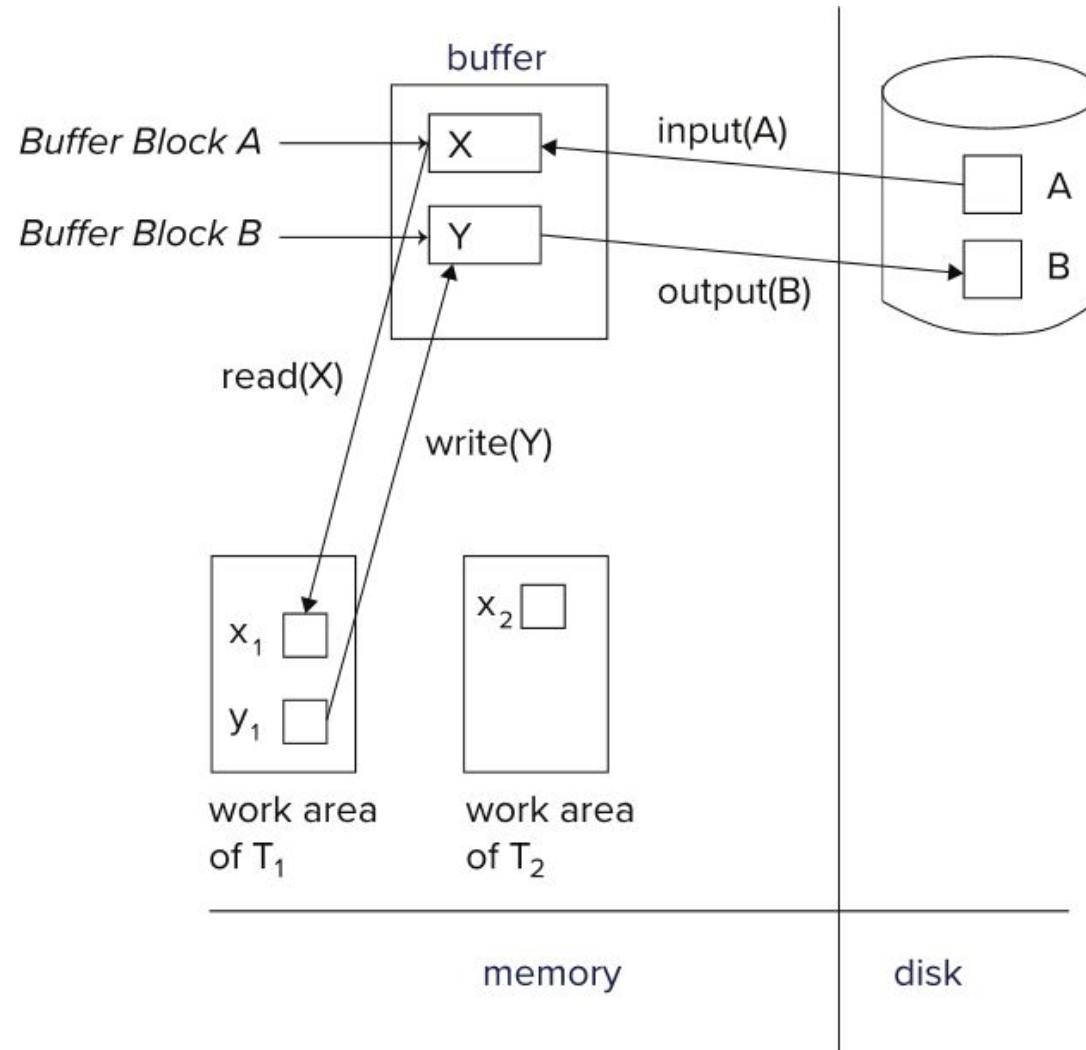


Storage Structure

- **Volatile storage:**
 - Does not survive system crashes
 - Examples: main memory, cache memory
- **Nonvolatile storage:**
 - Survives system crashes
 - Examples: disk, tape, flash memory, non-volatile RAM
 - But may still fail, losing data
- **Stable storage:**
 - A mythical form of storage that survives all failures
 - Approximated by maintaining multiple copies on distinct nonvolatile media
 - See book for more details on how to implement stable storage



Example of Data Access

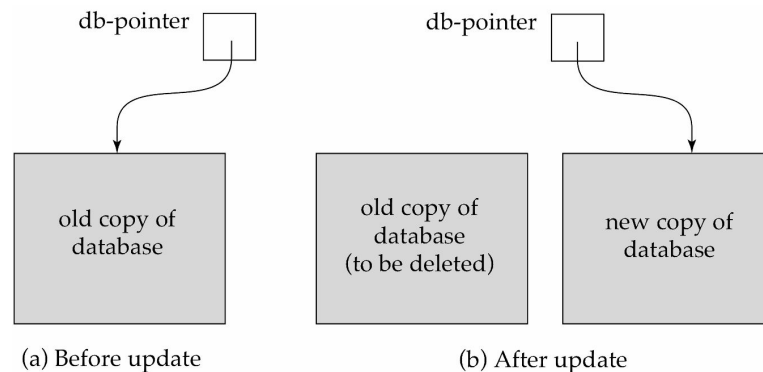


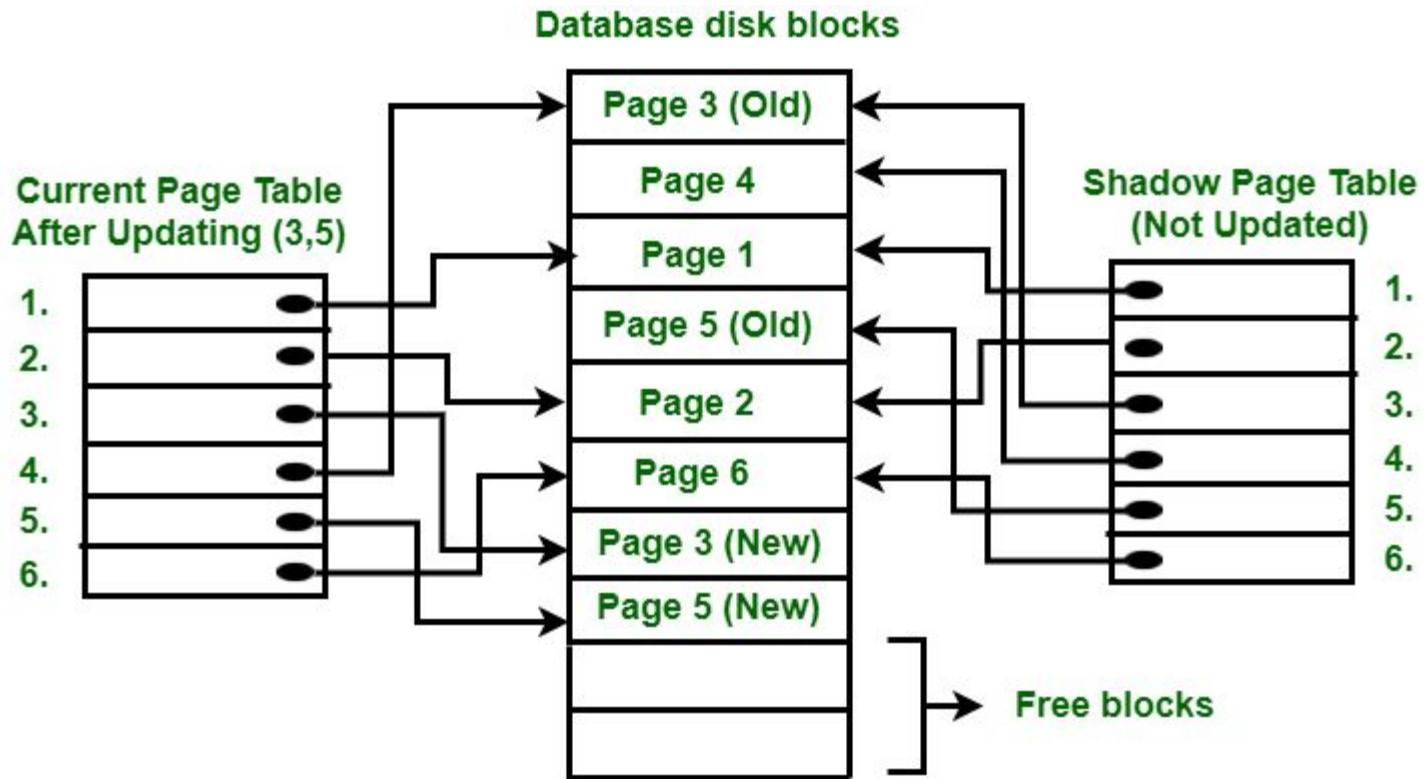


Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study **log-based recovery mechanisms** in detail
 - We first present key concepts
 - And then present the actual recovery algorithm
- Less used alternative: **shadow-copy** and **shadow-paging** (brief details in book)

shadow-copy







Log-Based Recovery

- A **log** is a sequence of **log records**. The records keep information about update activities on the database.
 - The **log** is kept on stable storage
- When transaction T_i starts, it registers itself by writing a $\langle T_i \text{ start} \rangle$ log record
- Before T_i executes **write**(X), a log record $\langle T_i, X, V_1, V_2 \rangle$ is written, where V_1 is the value of X before the write (the **old value**), and V_2 is the value to be written to X (the **new value**).
- When T_i finishes its last statement, the log record $\langle T_i \text{ commit} \rangle$ is written.
- Two approaches using logs
 - Immediate database modification
 - Deferred database modification.



Immediate Database Modification

- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
- Update log record must be written *before* database item is written
 - We assume that the log record is output directly to stable storage
 - (Will see later that how to postpone log record output to some extent)
- Output of updated blocks to disk can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.
- The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit
 - Simplifies some aspects of recovery
 - But has overhead of storing local copy



Transaction Commit

- A transaction is said to have committed when its commit log record is output to stable storage
 - All previous log records of the transaction must have been output already
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later



Immediate Database Modification Example

Log	Write	Output
$\langle T_0 \text{ start} \rangle$		
$\langle T_0, A, 1000, 950 \rangle$		
$\langle T_0, B, 2000, 2050 \rangle$		
	$A = 950$ $B = 2050$	
$\langle T_0 \text{ commit} \rangle$		
$\langle T_1 \text{ start} \rangle$		
$\langle T_1, C, 700, 600 \rangle$		
	$C = 600$	
$\langle T_1 \text{ commit} \rangle$		

B_B, B_C

B_A

B_B output after T_0
commits B_C
output before T_1
commits

B_A output after T_0
commits

- Note: B_X denotes block containing X .



Concurrency Control and Recovery

- With concurrent transactions, all transactions share a single disk buffer and a single log
 - A buffer block can have data items updated by one or more transactions
- We assume that *if a transaction T_i has modified an item, no other transaction can modify the same item until T_i has committed or aborted*
 - i.e., the updates of uncommitted transactions should not be visible to other transactions
 - Otherwise, how to perform undo if T_1 updates A, then T_2 updates A and commits, and finally T_1 has to abort?
 - Can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (strict two-phase locking)
- Log records of different transactions may be interspersed in the log.



Undo and Redo Operations

■ Undo and Redo of Transactions

- **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
 - Each time a data item X is restored to its old value V a special log record $\langle T_i, X, V \rangle$ is written out
 - When undo of a transaction is complete, a log record $\langle T_i \text{ abort} \rangle$ is written out.
- **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
 - No logging is done in this case



Recovering from Failure

- When recovering after failure:
 - Transaction T_i needs to be undone if the log
 - Contains the record $\langle T_i \text{ start} \rangle$,
 - But does not contain either the record $\langle T_i \text{ commit} \rangle$ or $\langle T_i \text{ abort} \rangle$.
 - Transaction T_i needs to be redone if the log
 - Contains the records $\langle T_i \text{ start} \rangle$
 - And contains the record $\langle T_i \text{ commit} \rangle$ or $\langle T_i \text{ abort} \rangle$



Recovering from Failure (Cont.)

- Suppose that transaction T_i was undone earlier and the $\langle T_i, \mathbf{abort} \rangle$ record was written to the log, and then a failure occurs,
- On recovery from failure transaction T_i is redone
 - Such a **redo** redoes all the original actions of transaction T_i *including the steps that restored old values*
 - Known as **repeating history**
 - Seems wasteful, but simplifies recovery greatly



Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

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

Recovery actions in each case above are.

- (a) undo (T_0): B is restored to 2000 and A to 1000, and log records $\langle T_0, B, 2000 \rangle$, $\langle T_0, A, 1000 \rangle$, $\langle T_0, \mathbf{abort} \rangle$ are written out
- (b) redo (T_0) and undo (T_1): A and B are set to 950 and 2050 and C is restored to 700. Log records $\langle T_1, C, 700 \rangle$, $\langle T_1, \mathbf{abort} \rangle$ are written out.
- (c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600



Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
 - Processing the entire log is time-consuming if the system has run for a long time
 - We might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing **checkpointing**
 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 L >** onto stable storage where L is a list of all transactions active at the time of checkpoint.
 4. All updates are stopped while doing checkpointing

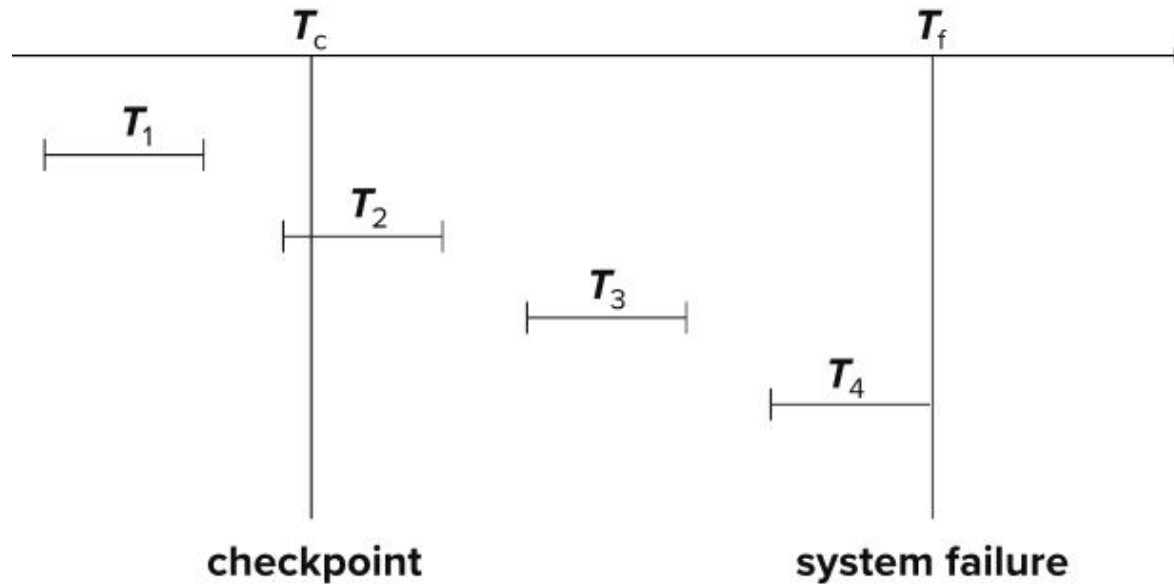


Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i .
 - Scan backwards from end of log to find the most recent **<checkpoint L >** record
 - Only transactions that are in L or started after the checkpoint need to be redone or undone
 - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.
- Some earlier part of the log may be needed for undo operations
 - Continue scanning backwards till a record **< T_i start>** is found for every transaction T_i in L .
 - Parts of log prior to earliest **< T_i start>** record above are not needed for recovery, and can be erased whenever desired.



Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone