

This document is for private circulation only, For
Third Year BTech. Computer 2020-21 CCOEW
students.

Subject: Database Management Systems (DMS)

Faculty: Prof. Jyoti Bangare

ACKNOWLEDGEMENT

- ❖ This presentation contains pictures, contents taken from multiple sources, authors and sites.
- ❖ We would like to appreciate and thank the authors, artists and do not claim any of following work to be ours, but it has been compiled here to use for academic purpose only.

Recovery System

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

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
 - 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 (multiple disks locally and at remote site)

Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
 - **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, for simplicity, that each data item fits in, and is stored inside, a single block.

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**
- Less used alternative: **shadow-paging**

Log-Based Recovery

- A **log** is kept on stable storage.
 - The log is a sequence of **log records**, and maintains a record of update activities on the database.
- 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
 - Deferred database modification
 - Immediate database modification

Immediate Database Modification

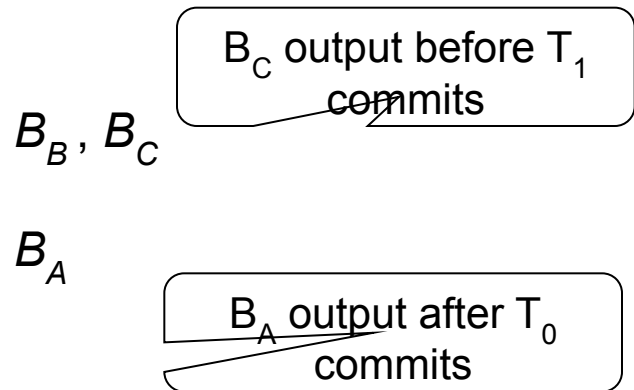
- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
- Update log record must be written *before* database item is written
 - We assume that the log record is output directly to stable storage
- Output of updated blocks to stable storage 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
 - Why allow outputs (writes to disk) to be postponed?
 - Reduces number of disk writes required to commit a transaction □ faster commit, earlier release of locks
 - Frequently updated pages will be written out to disk less often overall (multiple updates written out together).

Immediate Database Modification Example

Log	Write	Output
$\langle T_0 \text{ start} \rangle$		
$\langle T_0, A, 1000, 950 \rangle$		
$\langle T_0, B, 2000, 2050 \rangle$		
	$A = 950$ $B = 2050$	
$\langle T_0 \text{ commit} \rangle$		
$\langle T_1 \text{ start} \rangle$		
$\langle T_1, C, 700, 600 \rangle$		
	$C = 600$	
$\langle T_1 \text{ commit} \rangle$		
<ul style="list-style-type: none"> 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 T1 updates A, then T2 updates A and commits, and finally T1 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 kept in the log.

Undo and Redo Operations

- **Undo** of a log record $\langle T_i, X, V_1, V_2 \rangle$ writes the **old** value V_1 to X
- **Redo** of a log record $\langle T_i, X, V_1, V_2 \rangle$ writes the **new** value V_2 to X
- **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

Undo and Redo on 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 a $\langle T_i \text{ commit} \rangle$ or a $\langle T_i \text{ abort} \rangle$ record
 - 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 $\langle T_i \text{ abort} \rangle$
- Note that If transaction T_i was undone earlier and the $\langle T_i \text{ abort} \rangle$ record written to the log, and then a failure occurs, on recovery from failure T_i is redone
 - **such a redo redoes all the original actions *including the steps that restored old values***
 - Known as **repeating history**

Immediate DB Modification Recovery Example

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

$\langle T_0 \text{ start} \rangle$
 $\langle T_0, A, 1000, 950 \rangle$
 $\langle T_0, B, 2000, 2050 \rangle$

(a)

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

(b)

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

(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
 1. processing the entire log is time-consuming if the system has run for a long time
 2. we might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing **checkpointing**
 - recovery only needs to look at parts of the log after the checkpoint
 - plus just a little bit before the checkpoint corresponding to transactions that were active at the time of checkpoint.

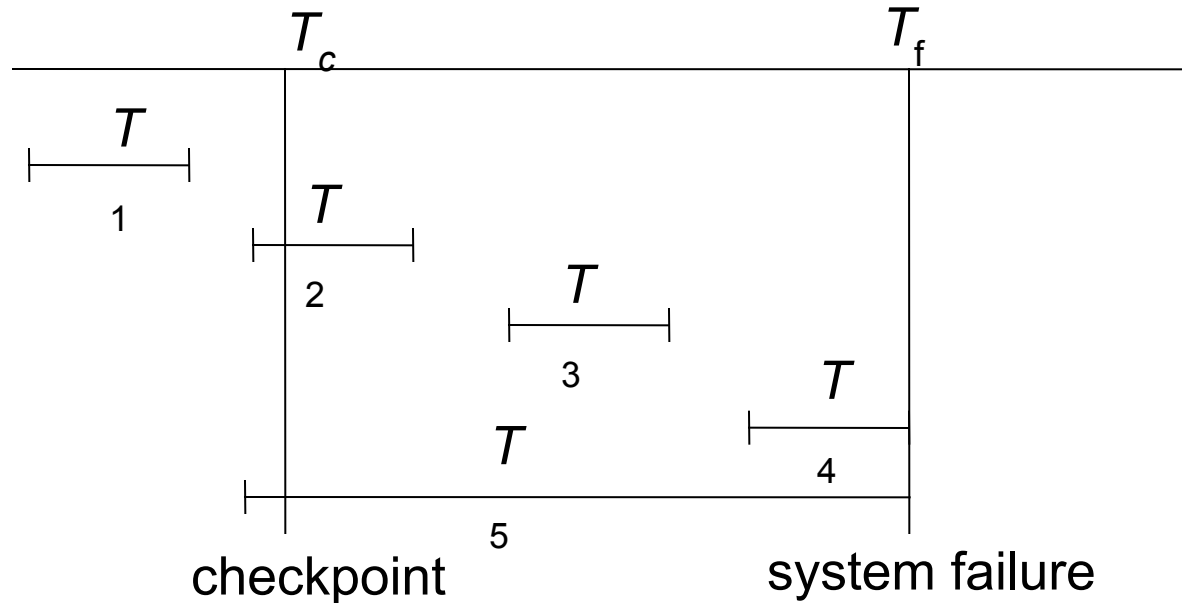
Checkpoints

- **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.
 - All updates are stopped while doing checkpointing
- (A less intrusive version of checkpointing is described in book)

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 .
 1. 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
 1. 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 (steps after checkpoint only) and T_3 redone.
- T_4 , T_5 undone

Recovery Algorithm

- **Logging** (during normal operation):
 - $\langle T_i \text{ start} \rangle$ at transaction start
 - $\langle T_i, X_j, V_1, V_2 \rangle$ for each update, and
 - $\langle T_i \text{ commit} \rangle$ at transaction end
- **Transaction rollback (during normal operation)**
 - Let T_i be the transaction to be rolled back
 - Scan log backwards from the end, and for each log record of T_i of the form $\langle T_i, X_j, V_1, V_2 \rangle$
 - perform the undo by writing V_1 to X_j
 - write a log record $\langle T_i, X_j, V_1 \rangle$
 - such log records are called **compensation log records**
 - Once the record $\langle T_i \text{ start} \rangle$ is found stop the scan and write the log record $\langle T_i \text{ abort} \rangle$

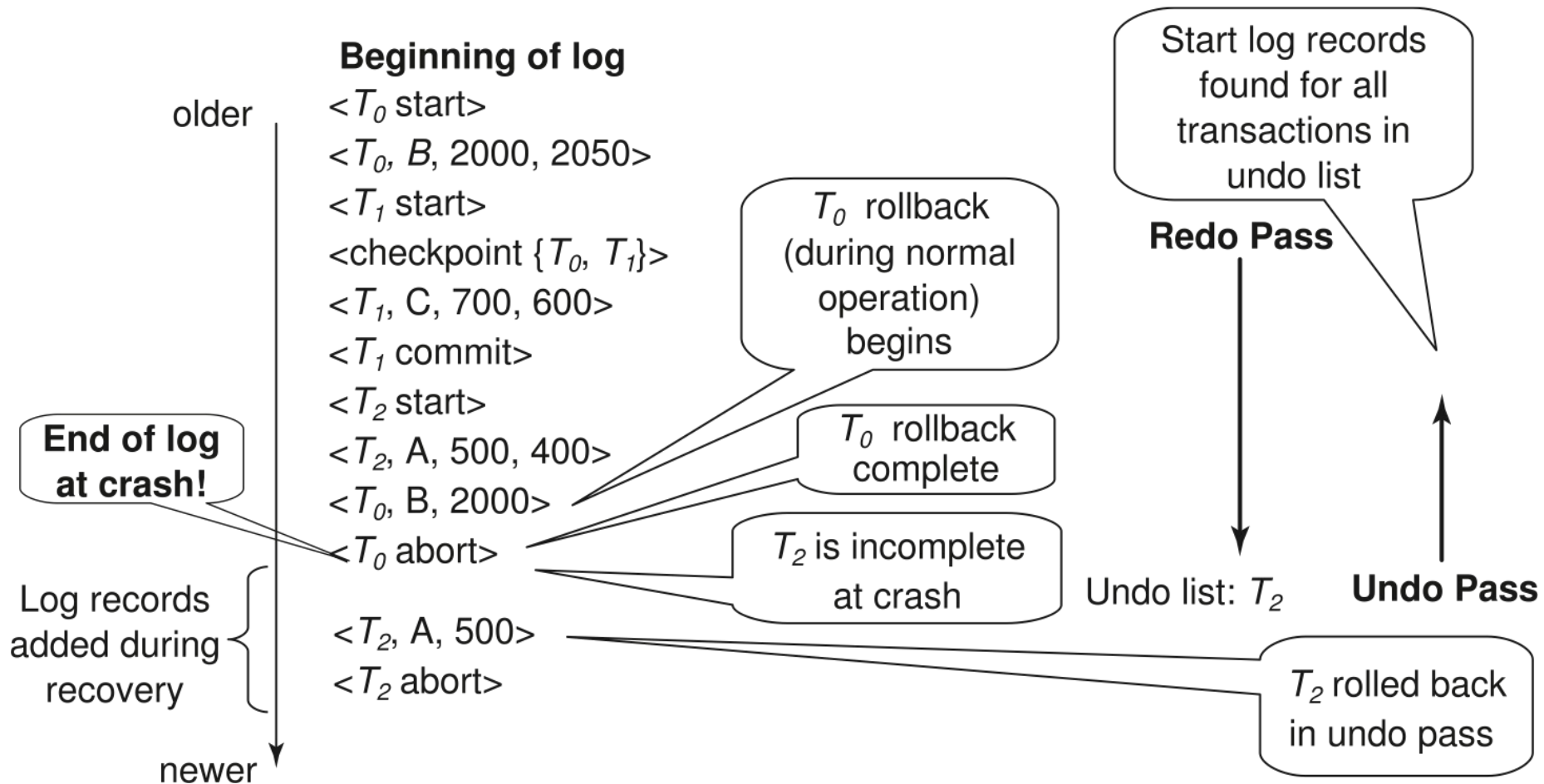
Recovery Algorithm (Cont.)

- **Recovery from failure:** Two phases
 - **Redo phase:** replay updates of **all** transactions, whether they committed, aborted, or are incomplete
 - **Undo phase:** undo all incomplete transactions
- **Redo phase:**
 1. Find last **<checkpoint L>** record, and set undo-list to L .
 2. Scan forward from above **<checkpoint L>** record
 1. Whenever a record $\langle T_i, X_j, V_1, V_2 \rangle$ is found, redo it by writing V_2 to X_j
 2. Whenever a log record $\langle T_i, \text{start} \rangle$ is found, add T_i to undo-list
 3. Whenever a log record $\langle T_i, \text{commit} \rangle$ or $\langle T_i, \text{abort} \rangle$ is found, remove T_i from undo-list

Recovery Algorithm (Cont.)

- **Undo phase:**
 1. Scan log backwards from end
 1. Whenever a log record $\langle T_i, X_j, V_1, V_2 \rangle$ is found where T_i is in undo-list perform same actions as for transaction rollback:
 1. perform undo by writing V_1 to X_j .
 2. write a log record $\langle T_i, X_j, V_1 \rangle$
 2. Whenever a log record $\langle T_i, \text{start} \rangle$ is found where T_i is in undo-list,
 1. Write a log record $\langle T_i, \text{abort} \rangle$
 2. Remove T_i from undo-list
 3. Stop when undo-list is empty
 - i.e. $\langle T_i, \text{start} \rangle$ has been found for every transaction in undo-list
 - After undo phase completes, normal transaction processing can commence

Example of Recovery



Shadow Paging

- **Shadow paging** is an alternative to log-based recovery; this scheme is useful if transactions execute serially
- Idea: maintain *two* page tables during the lifetime of a transaction –the **current page table**, and the **shadow page table**
- Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
 - **Shadow page table is never modified during execution**
- To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.
- Whenever any page is about to be written for the first time
 - **A copy of this page is made onto an unused page.**
 - **The current page table is then made to point to the copy**
 - **The update is performed on the copy**

Shadow Paging (Cont.)

Shadow Paging

- To manage access of data items by concurrent transactions two directories (current and shadow) are used.
 - The directory arrangement is illustrated below. Here a page is a data item.

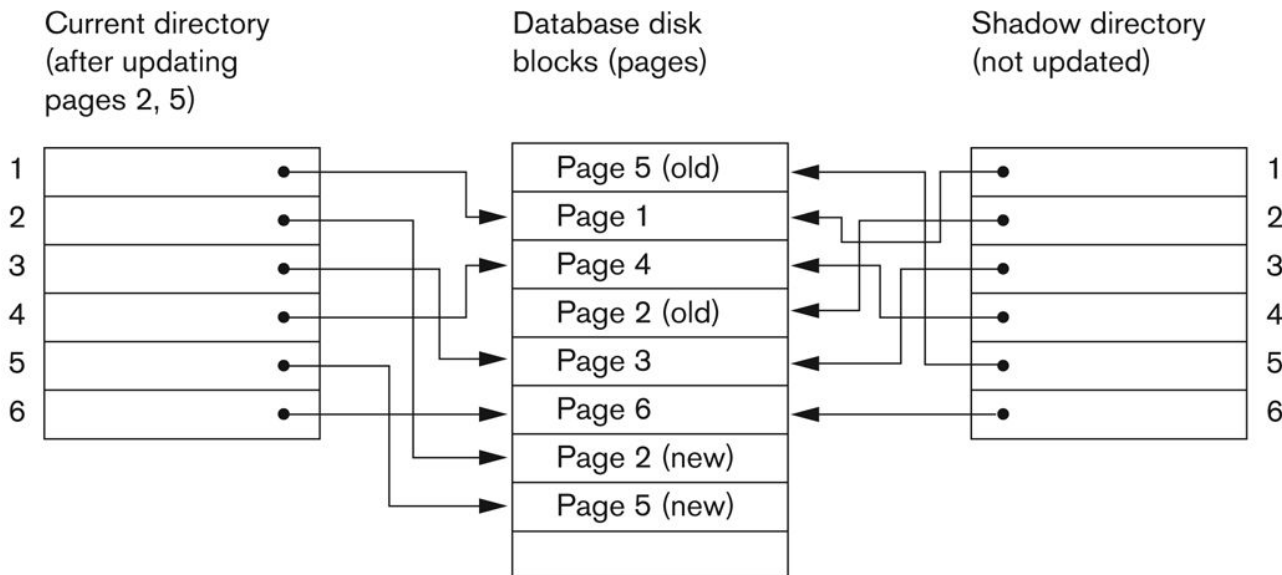


Figure 19.5
An example of shadow paging.

Shadow Paging (Cont.)

- To commit a transaction :
 1. Flush all modified pages in main memory to disk
 2. Output current page table to disk
 3. Make the current page table the new shadow page table, as follows:
 - keep a pointer to the shadow page table at a fixed (known) location on disk.
 - to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
- Once pointer to shadow page table has been written, transaction is committed.
- No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
- Pages not pointed to from current/shadow page table should be freed (garbage collected).

Shadow Paging

- Advantages of shadow-paging over log-based schemes
 - no overhead of writing log records
 - recovery is trivial
- Disadvantages :
 - Copying the entire page table is very expensive
 - Can be reduced by using a page table structured like a B⁺-tree
 - No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
 - Commit overhead is high even with above extension
 - Need to flush every updated page, and page table
 - Data gets fragmented (related pages get separated on disk)
 - After every transaction completion, the database pages containing old versions of modified data need to be garbage collected
 - Hard to extend algorithm to allow transactions to run concurrently
 - Easier to extend log based schemes