


CMPT 606


Read Chapters 22

Database Recovery Techniques

Dr. Abdelkarim Erradi

Department of Computer Science and Engineering

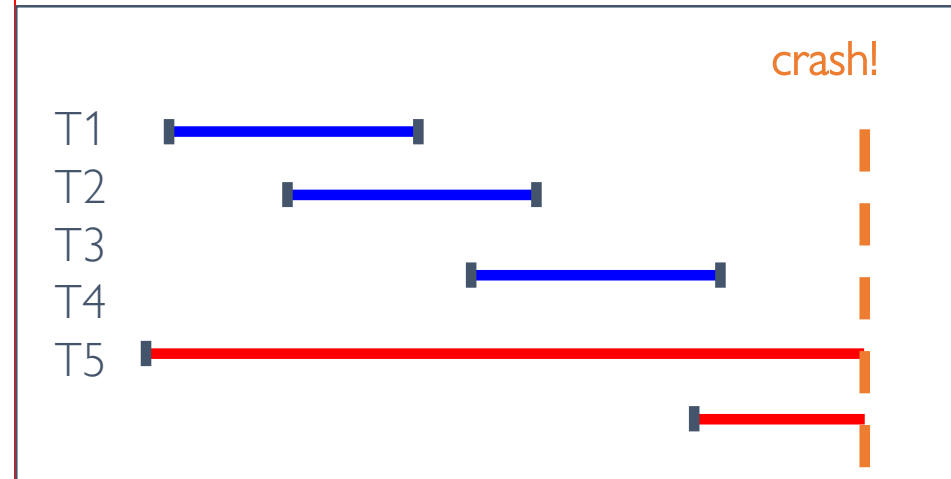
QU

Database Recovery

Motivation

- Purpose of database recovery is to ensure the transaction properties of:
 - **Atomicity:** Support the ability to rollback a transaction (via logging database changes i.e., updates, inserts and deletes)
 - **Durability:** Bring the database into the last consistent state, which existed prior to the failure

- Desired Behavior after system restarts:
 - T1, T2 & T3 should be durable
 - T4 & T5 should be aborted (effects undone)



Recovery in DBMS

- **Example (to illustrate consistency issues that can be introduced by failures)**

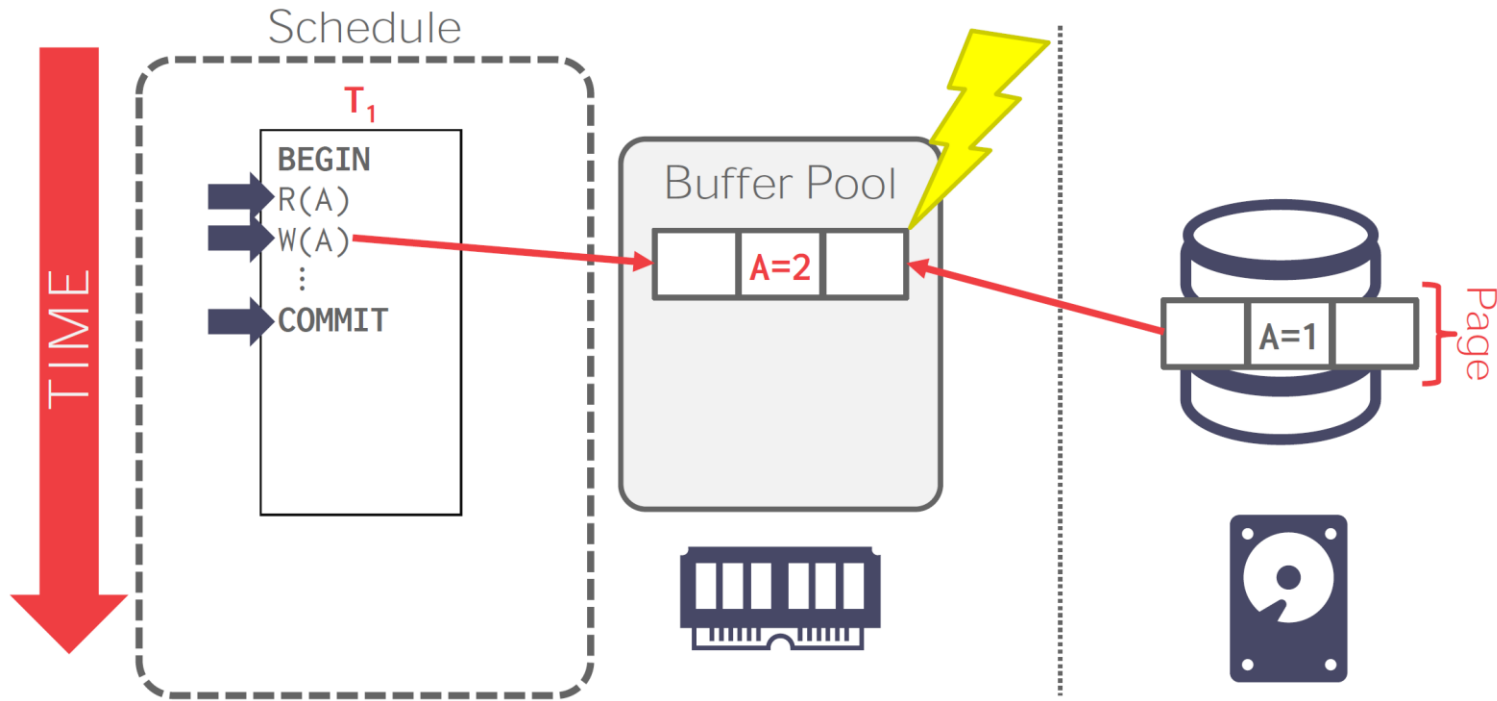
Bank transfer from account A to account B:

decrement the balance of account A by \$100;
increment the balance of account B by \$100;

- Scenario 1: Power failure after the first operation
 - **Such failures may leave database in an inconsistent state with partial updates carried out**
 - Transfer of funds from one account to another should either complete or not happen at all

=> Database logs come to the rescue to allow undoing incomplete transactions!

Why do we need the log?



- Data items to be modified are first read into Buffer Pool (in volatile memory)
- Updates are happening “**in place**” i.e. data is overwritten on (deleted from) its data Buffer.
- Modified Data Buffers are **later** flushed (written) back to the disk (**Deferred Update**). The disk version of the data item is overwritten by the Buffer version (**In-place update**) => Hence maintaining a log is critical for recovery

Challenge: REDO

Action	Buffer	Disk
Initially		0
T1 writes 1	1	0
T1 commits	1	0
CRASH		0

- Need to restore value 1 to item
 - Last value written by a committed transaction
- REDO: The process of **re-instating** the effects of a committed transaction for durability.

Challenge: UNDO

Action	Buffer	Disk
Initially		0
T1 writes 1	1	0
Page flushed		1
CRASH		1

- Need to restore value 0 to item
 - Last value from a committed transaction
- UNDO: The process of removing the effects of an incomplete or aborted transaction.

Transaction Log

- When **deferred in-place** update is used then log is necessary for recovery
- Data values prior to modification (BFIM - BeFore Image) and the new value after modification (AFIM – AFTer Image) are required
- These values are stored in a sequential file called Transaction log
- Log File records all write operations in the order in which they occur
- Is an append-only file

Current database state = current state in data file + log

Log file entries

Types of records (entries) in log file:

- [start_transaction, T]: Records that transaction T has started execution.
- [T, X, old_value, new_value]: T has changed the value of item X from old_value to new_value.
- [commit, T]: T has completed successfully, and committed.
- [abort, T]: T has been aborted.

Log file entries (Cont.)

- For **write_item** log entry, *old value* of item before modification (**Before Value used UNDO**) and the *new value* after modification (**After Value used REDO**) are stored.

A sample log:

```
< START T1 >  
<T1, A, 50, 25>  
<T1, B, 250, 25>  
<START T2>  
<T1, A, 75, 50>  
<T2, C, 35, 25>  
<COMMIT T1>  
<START T3>  
<T3, E, 55, 25>  
<T2, D, 45, 25>
```

Write-Ahead Logging (WAL)

- Maintain a log file separate from data files that contains the changes that txns make to database
 - Assume that the log is on stable storage.
 - Log contains enough information to perform the necessary undo and redo actions to restore the database.
- DBMS must write to disk the log file records that correspond to changes made to a database object **before** it can flush that object to disk.

Write-Ahead Logging (WAL)

The **Write-Ahead Logging** Protocol:

1. Must **force** the **log record** for an update **before** the corresponding data page gets to disk
2. Must **write all log records** for transaction **before commit returns** an acknowledgement to application:

A txn is not considered committed until all its log records have been written to disk.

- Property 1 guarantees Atomicity
- Property 2 guarantees Durability

Undo/Redo Logging

Update X  $\langle T_i, X, \text{old } X \text{ value}, \text{New } X \text{ value} \rangle$

- Undo/Redo Logging Rules

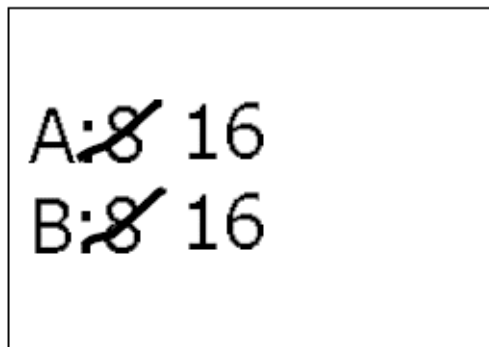
(1) Element X can be flushed (to data file) before or after T_i commit

(2) Before modifying X on disk, all corresponding log records should be flushed to disk

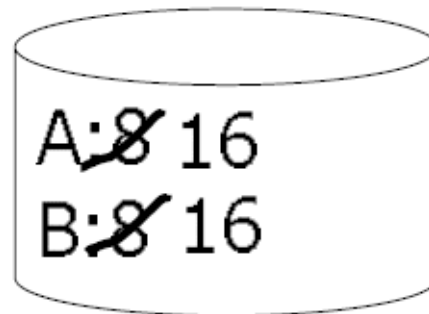
(3) Flush log before commit

Example

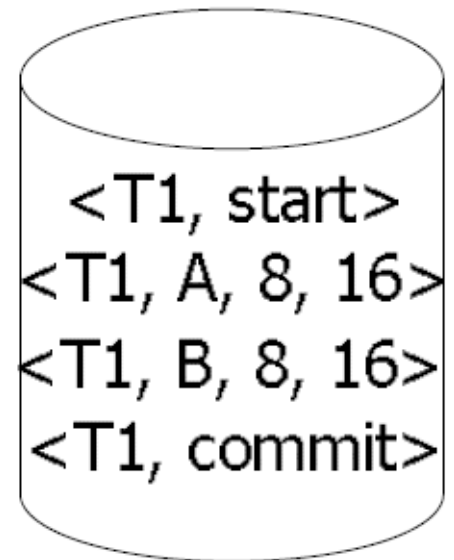
T₁: Read (A,t); $t \leftarrow t \times 2$ A=B
 Write (A,t);
 Read (B,t); $t \leftarrow t \times 2$
 Write (B,t);
 Output (A);
 Output (B);



memory



disk



log

Recovery Process

- The undo/redo recovery policy
 - Undo uncommitted transactions
 - Redo committed transactions
- **Backward pass** (end of log to the start)
 - Construct set **C** of transactions that committed
 - Undo all actions of transactions not in **C**
- **Forward pass** (start of log to the end)
 - Redo all actions of transactions in **C**

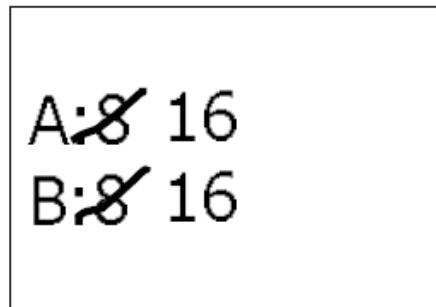
UNDO, REDO RECOVERY ACTIONS

- **Undo:** Restore all BFIMs from log to database on disk. UNDO proceeds backward in log
 - UNDO (roll-back) is needed for transactions that are not committed yet
- **Redo:** Restore all AFIMs from log to database on disk. REDO proceeds forward in log
 - REDO (roll-forward) is needed for committed transactions whose writes may have not yet been flushed from Buffer to Disk

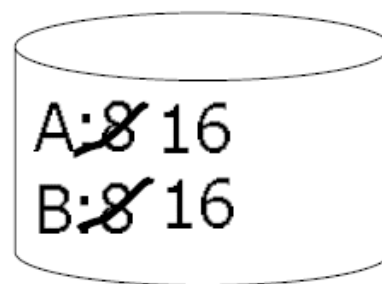
Recovery Example 1

T₁: Read (A,t); $t \leftarrow t \times 2$ A=B
 Write (A,t);
 Read (B,t); $t \leftarrow t \times 2$
 Write (B,t);
 Output (A);
 Output (B);

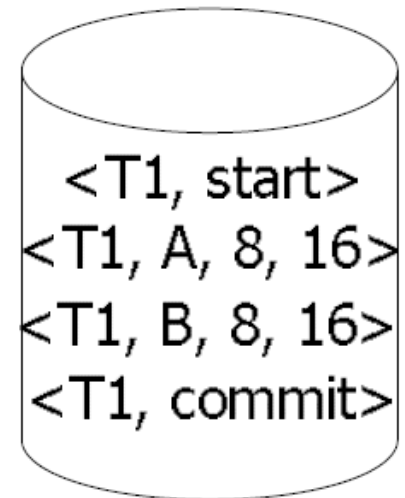
failure!



memory



disk

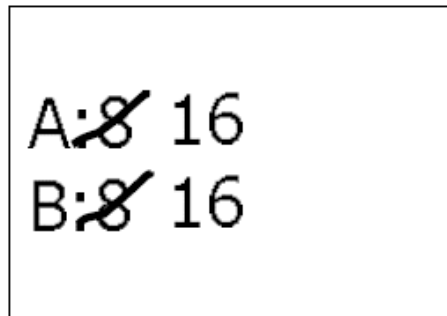


log

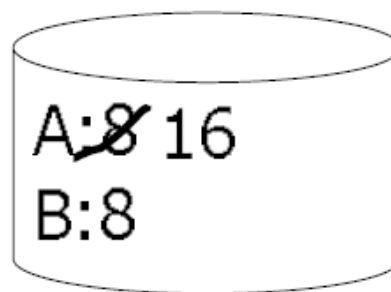
Recovery: T₁ is committed. Redo: write 16 to both A and B on disk.

Recovery Example 2

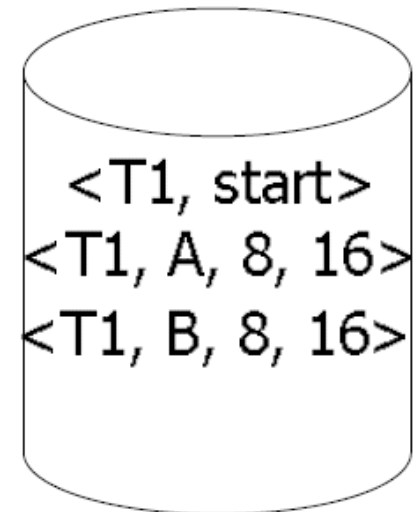
T₁: Read (A,t); $t \leftarrow t \times 2$ A=B
 Write (A,t);
 Read (B,t); $t \leftarrow t \times 2$
 Write (B,t);
~~Output (A);~~
 Output (B); failure!



memory



disk



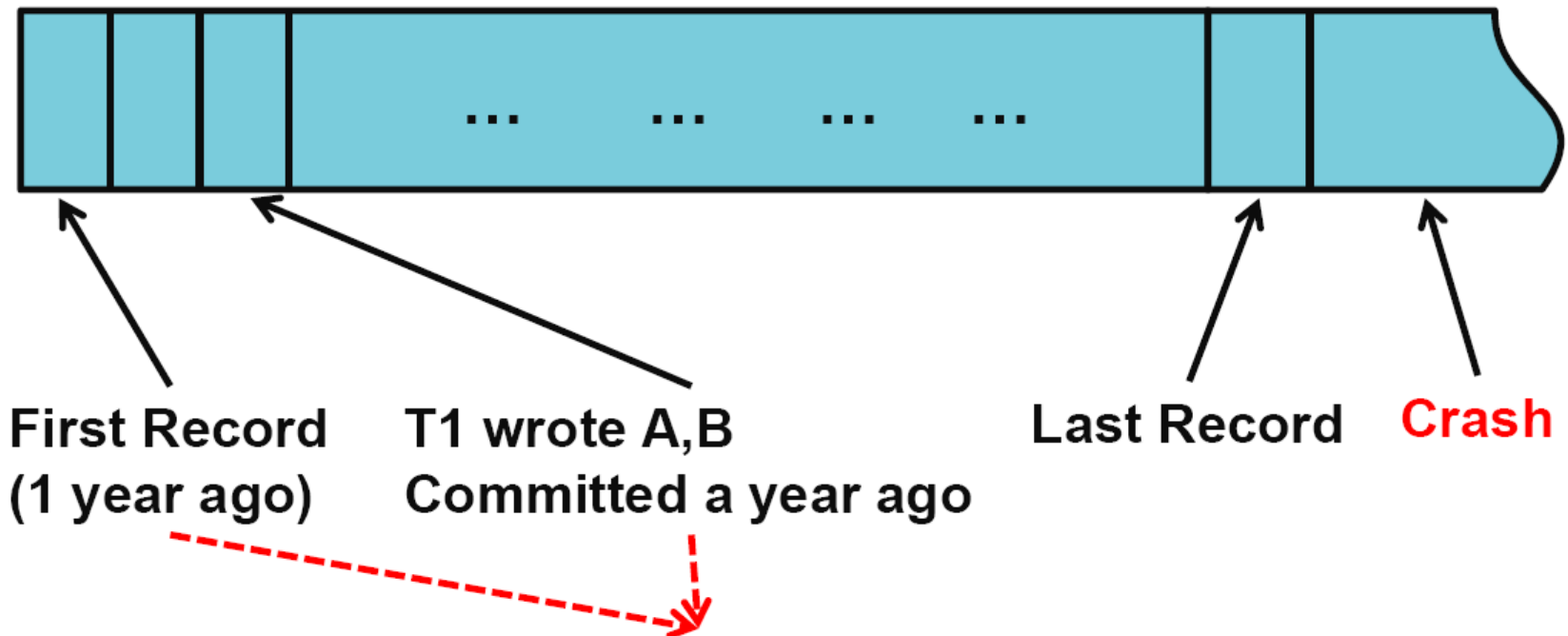
log

Recovery: T₁ incomplete. Undo: old values ("8") of A and B are written to disk. Write <T₁, abort> to the log.

Recovery is SLOW!

- Recovery can be slow if the log file is big. Hence Checkpointing is used to reduce recovery time

A Redo Log:

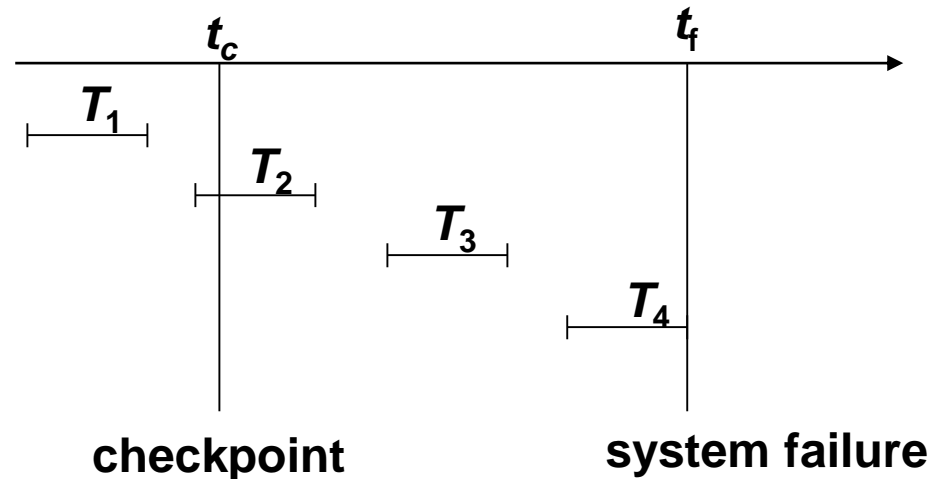


Still, need them to redo after crash!!

Checkpointing

- From time to time the database flushes its buffer to database disk to minimize the task of recovery.
- Following steps defines a checkpoint operation:
 1. Suspend execution of transactions temporarily
 2. Write "**start checkpoint**" listing all active transactions to log
 3. Force write dirty Data Buffers to disk whether or not their transactions have committed
 4. Write "end checkpoint" to log, save the log to disk.
 5. Resume normal transaction execution.
- During recovery redo or undo is required to transactions appearing after [checkpoint] record

Checkpointing Contd..



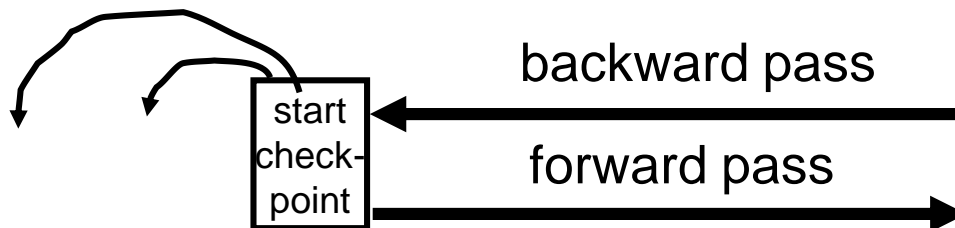
- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone
- T_4 undone

End Checkpoint

- If the system successfully performed the checkpointing (i.e., wrote **END CHECKPOINT** log entry to disk) then this means that all Dirty Data Buffers before the **START CHECKPOINT** must have been forced written to disk
 - If the crash happens after the **END CHECKPOINT** then we only redo actions of completed transactions after START CKPT (NOT from the start of the log file)
- This is basically the benefit of checkpointing i.e., avoid the need to redo ALL successful actions from the beginning of the log in order to reduce the recovery time

Recovery process

- Backwards pass (end of log to most recent checkpoint start)
 - construct set **S** of committed transactions
 - undo actions of transactions not in **S**
- Forward pass (latest checkpoint start to end of log)
 - redo actions of **S** transactions



Recovery Example 3

Redo/Undo Log:

<T1, START>

<T1, A, 4,5>

<T2, START>

<T1, COMMIT>

<T2, B, 9, 10>

<START CKPT(T2)>

<T2, C, 14, 15>

<T3, START>

<T3, D, 19, 20>

<END CKPT>

<T2, COMMIT>

<T3, COMMIT>

- T2 and T3 already committed.
- Since we see <END CKPT> first (backward), T1's changes must be on disk. So T1 can be ignored.
- Redo T2 and T3, forward.
 - Do NOT need to look at the log records before <START CKPT>
 - Reason: their changes are already on disk

Recovery Example 4

Redo/Undo Log:

<T1, START>

<T1, A, 4,5>

<T2, START>

<T1, COMMIT>

<T2, B, 9, 10>

<START CKPT(T2)>

<T2, C, 14, 15>

<T3, START>

<T3, D, 19, 20>

<END CKPT>

<T2, COMMIT>

- T2 committed.
- T3 incomplete
- Redo T2 (forward): C = 15 (on disk)
- Undo T3 (backward): D = 19 (on disk)

Recovery Example 5

Redo/Undo Log:

<T1, START>

<T1, A, 4,5>

→ **<T2, START>**

<T1, COMMIT>

<T2, B, 9, 10>

<START CKPT(T2)>

<T2, C, 14, 15>

<T3, START>

<T3, D, 19, 20>

<END CKPT>

- T2 and T3 incomplete
- Undo T3 (backward): D = 19 (on disk)
- Undo T2 (backward): C = 14 and B = 9 (on disk)

Recovery Example 6

Redo/Undo Log:

→ <T1, START>
<T1, A, 4, 5>
<T2, START>
<T1, COMMIT>
<T2, B, 9, 10>
<START CKPT(T2)>
<T2, C, 14, 15>
<T3, START>
<T3, D, 19, 20>

- T2 active and incomplete.
 - Undo (backward): C = 14 and B = 9 (on disk)
- T3 incomplete
 - Undo (backward): D = 19 (on disk)
- T1 complete
 - Redo (forward): A = 5 (on disk)

Summary

- **Recovery Manager uses** Write-Ahead Logging (WAL) to guarantee Atomicity & Durability
 - On recovery: undo uncommitted txns + redo committed txns
- **Checkpointing** speeds-up recovery by limiting the amount of log to scan on recovery
- Recovery from Media Failure using Database Backup + Log Backup
- Replication could be used for High Availability and Failover