Recovery: Outline

- Logging Mechanism
  - Motivation
  - Log-Based Recovery
- Recovery Algorithms
  - Undo/Redo
  - No-Undo/Redo
  - Undo/No-Redo
  - No-Undo/No-Redo
- Checkpoint
- Other Issues
  - Media Failure and Archiving
  - Recovery Tuning

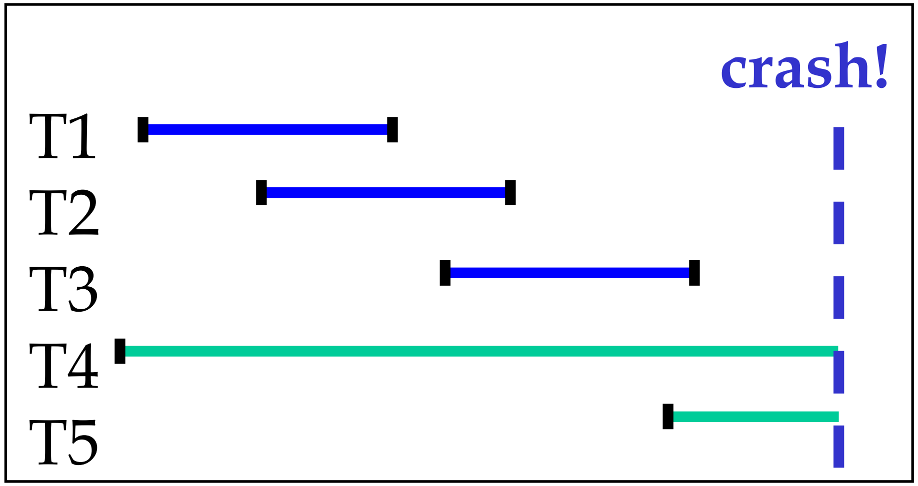
Recovery

- Logging Mechanism
  - Motivation
  - Log-Based Recovery
- Recovery Algorithms
- Checkpoint
- Other Issues

Motivation

- Atomicity:
  - Transactions may abort ("Rollback").
- Durability:
  - What if DBMS stops running? (Causes?)

❖ Desired Behavior after system restarts:
  – T1, T2 & T3 should be durable.
  – T4 & T5 should be aborted (effects not seen).



Atomicity and Durability of Operations

- Each database operation
  - either completes fully,
  - or does not happen at all.
- Low-level atomic operations are built into hardware.
- High-level atomic operations are called *transactions*.
- The DBMS ensures that a transaction
  - either completes and has a permanent result (*committed* transaction),
  - or has no effect at all on the database (*aborted* transaction).
- The role of recovery is to ensure atomicity and durability of transactions in the presence of system failures.

A Sample Transaction

- Transfer $50 from account A to account B.
  1 Transaction starts
  2 Read A
  3 A ← A - 50
  4 Write A
  5 Read B
  6 B ← B + 50
  7 Write B
  8 Transaction ends
- If initial values are A = 180, B = 100, then after the execution A = 130 and B = 150.
- Sample values of database variables at various points in execution:

Last Instruction	A	B
2	180	100
5	130	100
7	130	150

Problems In Ensuring Atomicity and Durability

- *Logical errors*: A transaction cannot complete due to some internal condition, e.g., bad input, overflow, resource limit exceeded.
- *System errors*: The database system must terminate an active transaction due to, e.g., deadlock. The transaction can be executed later.
- *System crash*: A power failure or other hardware failure causes the system to crash. Volatile storage is lost, non-volatile is not.
- *Disk failure*: A head crash or similar failure destroys all or part of disk storage (failure during data transfer, fire, theft, sabotage, mounting wrong disk, etc.). Nonvolatile storage is lost.

Recovery

- Initial (incorrect) procedures
  - Re-execute the transaction.
    - If the sample transaction crashed after instruction 4 (or later), incorrect values (A=130,...) will exist in the database.
  - Do not re-execute the transaction.
    - If the sample transaction crashed before instruction 7, incorrect values (..., B = 100) will exist in the database.
- Problems
  - Which instruction was last executed?
  - Multiple, concurrent users
  - Buffer management

A = 180, B = 100

1 Transaction starts
2 Read A
3 A ← A - 50
4 Write A
5 Read B
6 B ← B + 50
7 Write B
8 Transaction ends

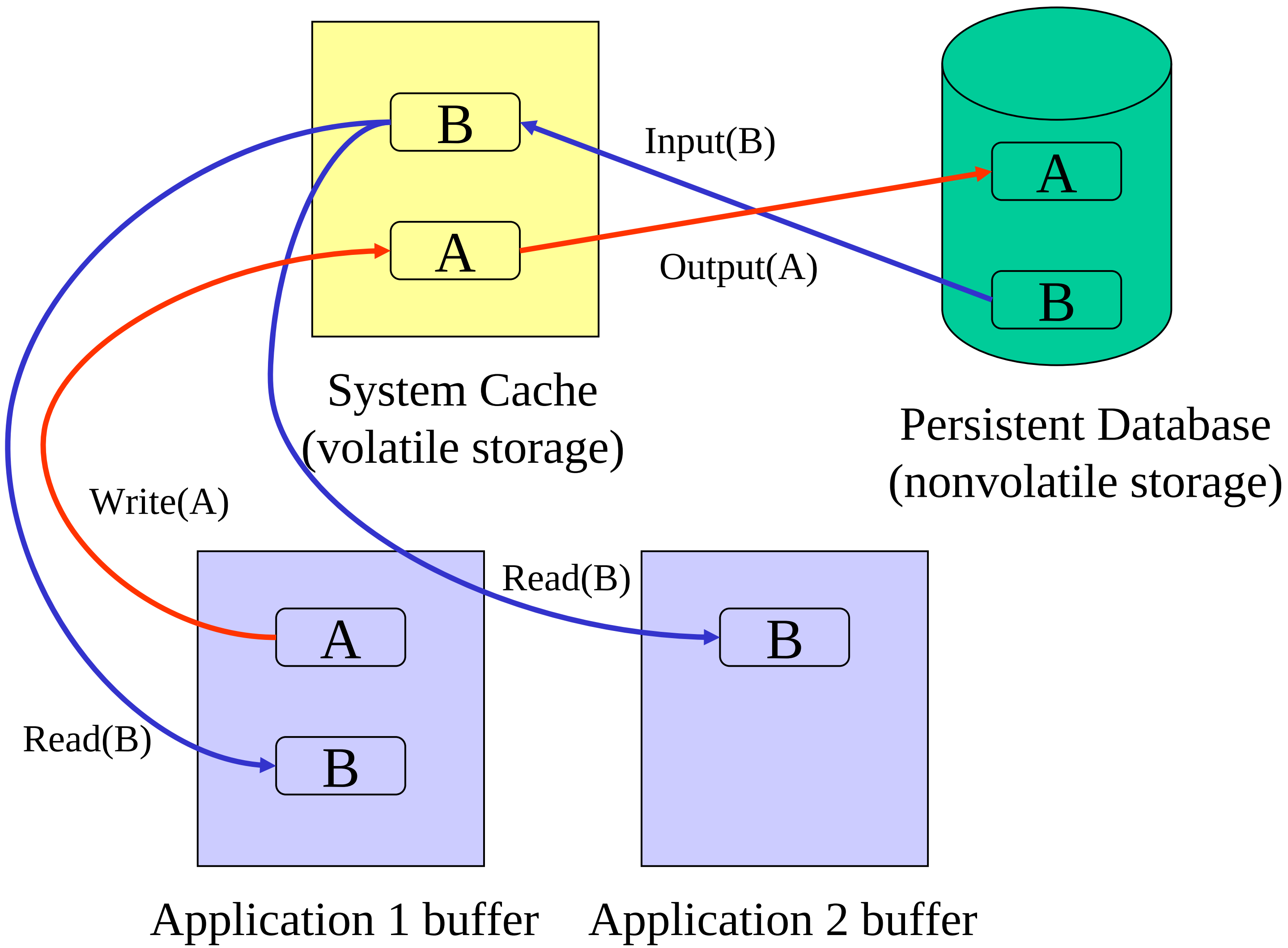
Storage Types

- Management of failure takes advantage of several types of storage.
- *Volatile storage*: does not survive system crashes
  - Main memory
  - Cache memory
- *Non-volatile storage*: survives system crashes
  - Disk
  - Tape
- *Stable storage*: a mythical form of storage that survives all failures
  - Approximated by maintaining multiple copies of distinct non-volatile media with independent failure modes.

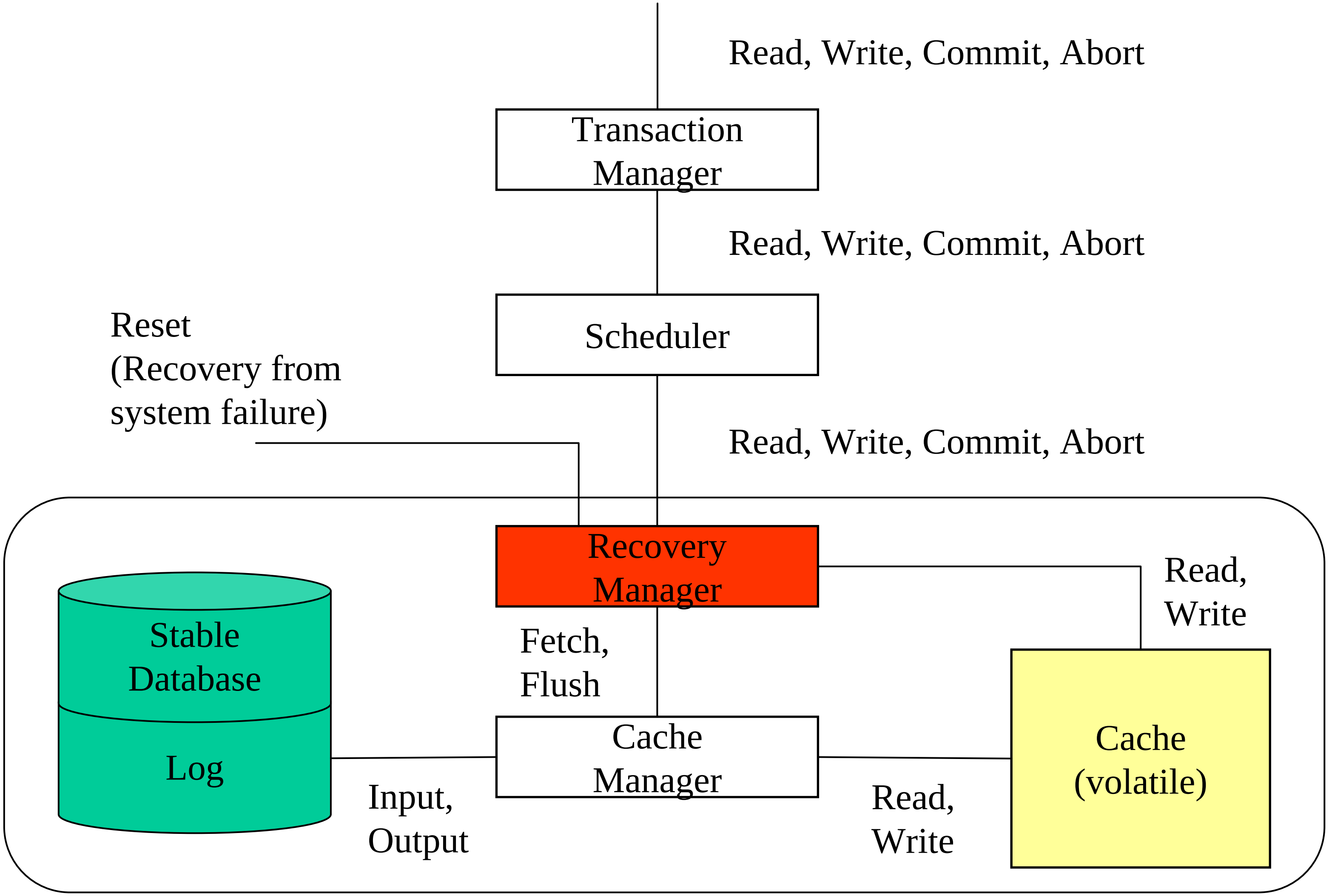
Storage Operations

- Transactions access and update the database.
- There are two operations for moving blocks with data items between disk and main memory.
  - Input(Q) transfers the block containing data item Q to main memory.
  - Output(Q) transfers the block containing Q to disk and replaces the appropriate block there.
- There are two operations for moving values between data items and application variables.
  - Read(Q,q) assigns the value of data item Q to variable q.
  - Write(Q,q) assigns the value of variable q to data item Q.
    - An Input(Q) may be necessary for both operations.

Movement of Values



System Model



Log-based Recovery

- During normal operation, the following occurs.
- Each transaction T when it starts, registers itself on the log, by outputing <T starts>.
- When a transaction T modifies a data item X by Write(X), then the Recovery Manager first outputs an entry with the following fields to the log.
  - transaction's ID (i.e., T)
  - data item name (i.e., X)
  - old value of the item
  - new value of the item
- Then the Recovery Manager writes the new value of X into the cache.

Log-based Recovery, cont.

- Later, the Cache Manager can output the value to the portion of the database on disk.
- This allows output to stable storage to be asynchronous with application writes, which is especially effective in the presence of large main memories.
- When T reaches its last statement, the Recovert Manager appends <T commits> to the log, and T then commits.
  - The transaction commits precisely when the commit entry (after all previous entries for this transaction) is outputted to the log.

Buffering of Log Entries and Data Items

- We have so far assumed that log entries are put on stable storage when they are created.
- Lifting this restriction, but imposing the following, leads to better performance.
  - Transaction T cannot commit before <T commits> has been output to stable storage.
  - <T commits> cannot be placed on stable storage before all other log entries pertaining to T are on stable storage.
  - A block of data items cannot be output to stable storage until all log entries pertaining to the data items are output.

Recovery From Failures

- When failures occur, the following operations that use the log are executed.
  - *Undo*: restore database to state prior to execution.
  - *Redo*: perform the changes to the database over again.

- Examples
  - Undo
    - Aborted transaction
    - Active transactions at time of crash
  - Redo
    - Committed transaction whose changes have not been placed in the database

Example

- We consider two transactions executed sequentially by the system.
  - T1:

T1	T2
Read(A)	Read(A)
$A \leftarrow A + 50$	$A \leftarrow A + 10$
Read(B)	Write(A)
$B \leftarrow B + 100$	Read(D)
Write(B)	$D \leftarrow D - 10$
Read(C)	Read(E)
$C \leftarrow 2C$	Read(B)
Write(C)	$E \leftarrow E + B$
$A \leftarrow A + B + C$	Write(E)
Write (A)	$D \leftarrow D + E$
	Write(D)

- The initial values are:

  A=100, B=300, C=5, D=60, E=80

Example, cont.

- The Log
  1. <T1 starts>
  2. <T1, B, old: 300, new: 400>
  3. <T1, C, old: 5, new: 10>
  4. <T1, A, old: 100, new: 560>
  5. <T1 commits>
  6. <T2 starts>
  7. <T2, A, old: 560, new: 570>
  8. <T2, E, old: 80, new: 480>
  9. <T2, D, old: 60, new: 540>
  10. <T2 commits>

A=100 B=300 C=5 D=60 E=80

T1:	T2:
Read(A)	Read(A)
$A \leftarrow A + 50$	$A \leftarrow A + 10$
Read(B)	Write(A)
$B \leftarrow B + 100$	Read(D)
Write(B)	$D \leftarrow D - 10$
Read(C)	Read(E)
$C \leftarrow 2C$	Read(B)
Write(C)	$E \leftarrow E + B$
$A \leftarrow A + B + C$	Write(E)
Write (A)	$D \leftarrow D + E$
	Write(D)

- Output of B can occur anytime after entry 2 is output to the log, etc.

Example, cont.

- Assume a system crash occurs. The log is examined. Various actions are taken depending on the last instruction (actually) written on it.

1. <T1 starts>
2. <T1, B, …>
3. <T1, C, …>
4. <T1, A, …>
5. <T1 commits>
6. <T2 starts>
7. <T2, A, …>
8. <T2, E, …>
9. <T2, D, …>
10. <T2 commits>

Last Instruction	**Action**	**Consequence**
I = 0	Nothing	Neither T1 nor T2 has run
1 ≤ I ≤ 4	*Undo T1*: Restore the values of variables listed in 1-I old values	T1 has not run
5 ≤ I ≤ 9	*Redo T1*: Set the values of the variables listed in I-4 to values created by T1 *Undo T2*: Restore the values of variables listed in I-9 to those before T2 started execution	T1 ran T2 has not run
I=10	*Redo T1* *Redo T2*	T1 and T2 both ran

Recovery: Outline

- Logging Mechanism
- Recovery Algorithms
  - Undo/Redo
  - No-Undo/Redo
  - Undo/No-Redo
  - No-Undo/No-Redo
- Checkpoint
- Other Issues

The Undo/Redo Algorithm

- Following a failure, the following is done.
  - Redo all transactions for which the log has both “start” and “commit” entries.
  - Undo all transactions for which the log has “start” entry but no “commit” entry.
- Remarks:
  - In a multitasking system, more than one transaction may need to be undone.
  - If a system crashes during the recovery stage, the new recovery must still give correct results.
  - In this algorithm, a large number of transactions need to be redone, since we do not know what data items are on disk.

Undo/Redo, cont.

- Goal: Maximize efficiency during normal operation.
  - Some extra work is required during recovery time.
- Allows maximum flexibility of buffer manager.
  - Database outputs are entirely asynchronous, other than having to happen after the corresponding log entry outputs.
- Most complex at recovery time
  - Must implement both undo and redo.
- Also termed *immediate database modification*
- Note: requires before and after images in log or only after images if an initial before image is written to the log before first write.

No-Undo/Redo

- Algorithm
  - Don't output values to the disk until the commit log entry is on stable storage.
  - All writes go to the log and to the database cache.
  - Sometime after commit, the cached values are output to the disk.
- Advantages
  - Faster during recovery: no undo.
  - No before images needed in log.
- Disadvantages
  - Database outputs must wait.
  - Lots of extra work at commit time.
- Also called *deferred database modification*.

Undo/No-Redo

- Algorithm
  - All changed data items need to be output to the disk before commit.
    - Requires that the write entry first be output to the (stable) log.
  - At commit:
    - Output (*flush*) all changed data items in the cache.
    - Add commit entry to log.
- Advantages
  - No after images are needed in log.
  - No transactions need to be redone.
- Disadvantages
  - Hot spot data requires a flush for each committed write.
    - Implies lots of I/O traffic.

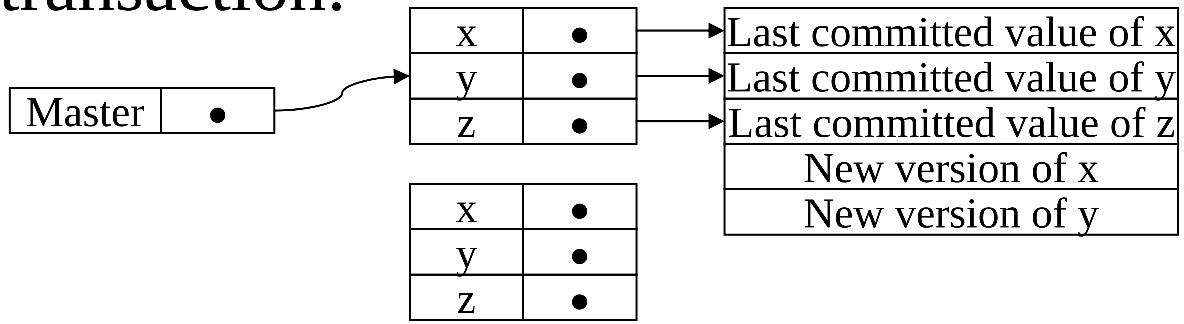
No-Undo/No-Redo

- Algorithm
  - No-undo $\rightarrow$ don't change the database during a transaction
  - No-redo $\rightarrow$ on commit, write changes to the database in a single atomic action
- Advantages
  - Recovery is instantaneous.
  - No recovery code need be written.

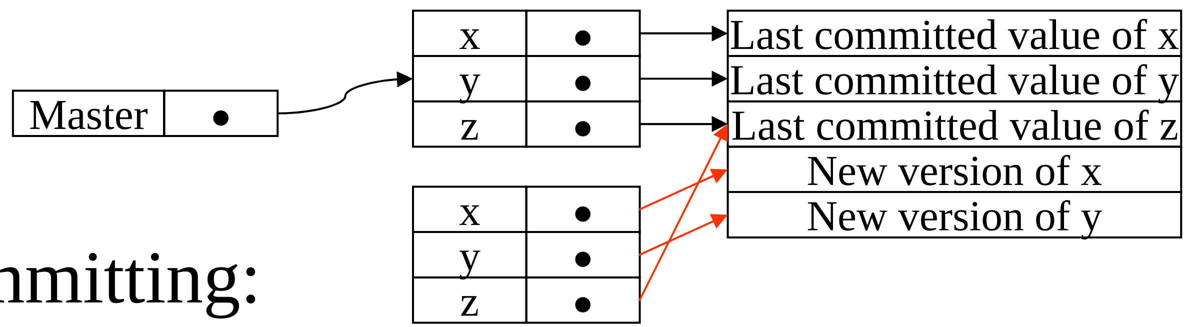
- Atomic databases writes of many pages is accomplished using the *shadow paging* technique.

No-Undo/No-Redo, cont.

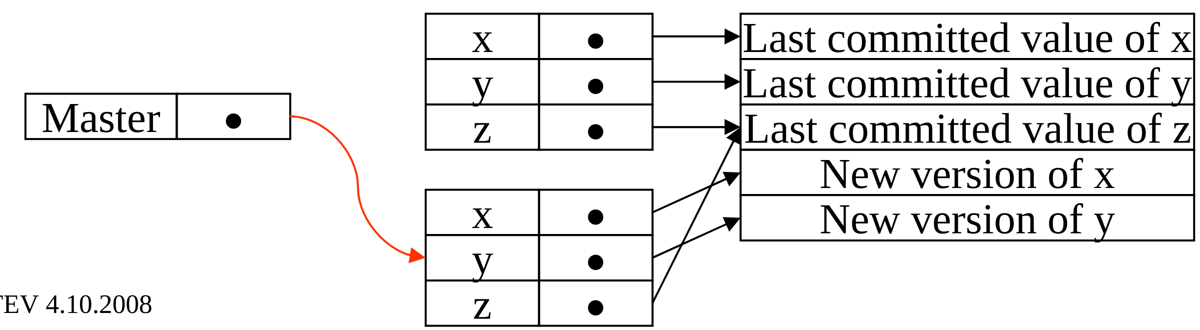
- During a transaction:



- After preparing new directory for commit:



- After committing:



No-Undo/No-Redo Example, cont.

- Commit requires writing on disk of one bit.
- Restart is very fast: one disk read.
- Problems:
  - Access to stable storage is indirect (though directories may be possibly stored in main memory).
  - Garbage collection of stable storage is required.
  - Original layout of data is destroyed.
  - Concurrent transactions are difficult to support.

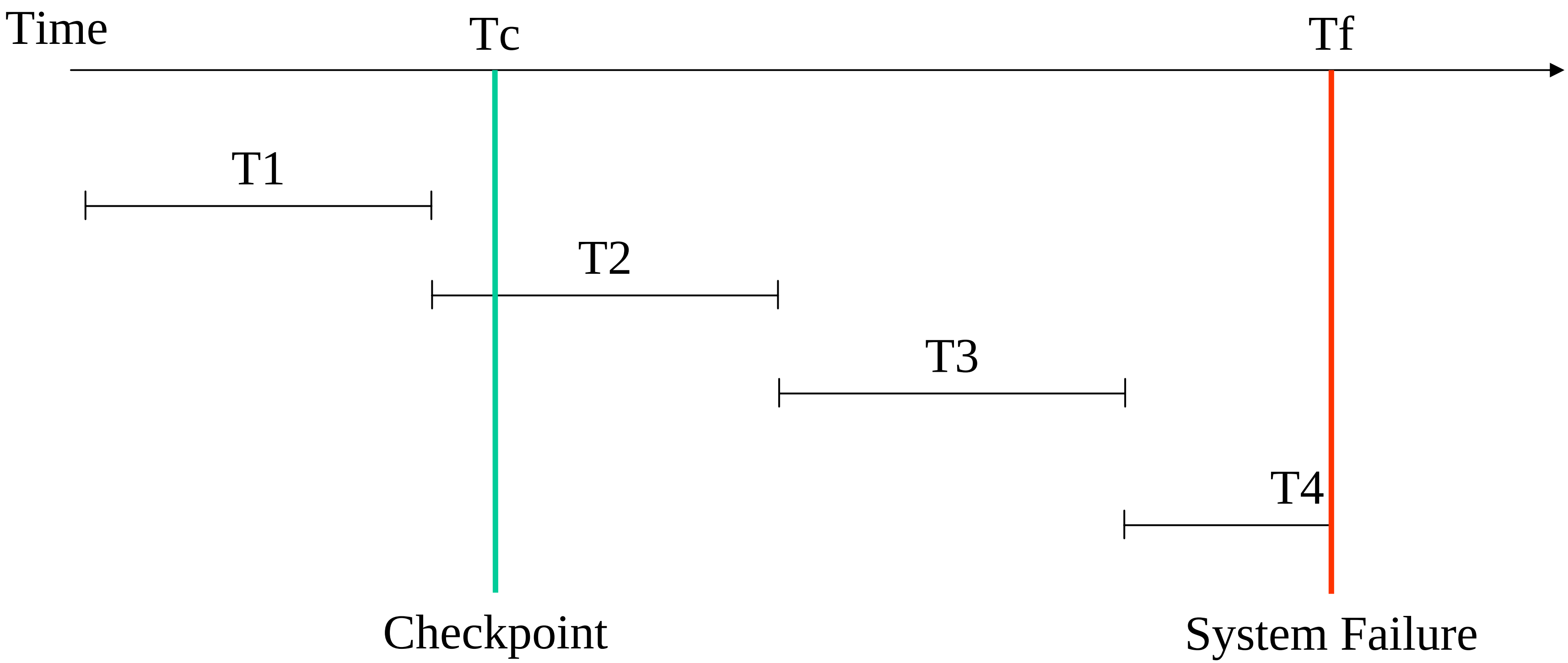
Recovery

- Logging Mechanism
- Recovery Algorithms
- Checkpoint
- Other Issues

Checkpoint

- Checkpoint speeds up recovery by flushing dirty pages to disk.
- During the execution in addition to the activities of the previous method, periodically perform checkpointing:
  1. Output the log buffers to the log.
  2. Force database buffers to the disk.
  3. Output an entry <checkpoint> on the log.
- During recovery
  - Undo all transactions that have not committed.
  - Redo all transactions that have committed *after* checkpoint.

Example - Recovery with Checkpoints



- T1 is ok.
- T2 and T3 are redone.
- T4 is undone.

Recovery

- Logging Mechanism
- Recovery Algorithms
- Checkpoint
- Other Issues
  - Media Failure and Archiving
  - Recovery Tuning

Media Failure

- Stable database or log may be wiped out.
  - Head crash
  - Operator error, e.g., mounting incorrect tape
  - Power lossage
  - Sabotage
  - Disgruntled employee
- Only solution: keep multiple copies on multiple devices with multiple failure rates.
- Mirroring
  - Each write applied to one disk, then to other.
  - On failure, use good device until bad device is fixed and current data restored.
    - During this time, there is no redundancy.
  - Increases the disk input rate, but not the disk output rate.

Archiving

- Protects against loss of non-volatile storage.
- Periodically dump entire database, in a consistent state, to an archive database.
- Mirror the logs.
- Recovery algorithms
  - Log destroyed: use other one; rebuild destroyed copy.
  - Stable database destroyed: apply log to archive.
  - Both logs destroyed: start again with archive.
    - Repeat lost transactions with information from paper records.
- Creating archive database can be expensive.
  - Many businesses are 24x7 operations.
  - Creating a consistent state is basically a huge read-only transaction.

Recovery Tuning

- Place the log on a separate disk.
  - This yields sequential log-writes and also improves reliability.
- Delay writes to the database disk(s).
  - Buffered database writes may reduce the random writes to the database disks.
  - Sometimes writes can be avoided completely (repeated updates of the same data item).
- Tune checkpoint and dump intervals.
  - Trade-off between performance of normal operation and performance of recovery
  - High-availability applications should do checkpoints very often, say every 20 minutes.

Interactive Transactions

- Interactive transactions are difficult to handle satisfactorily.
  - How to rollback a message to the user?
  - How to recall $100 an automatic teller handed out?
- Correct but unsatisfactory solutions:
  - Forbid interactive transactions.
  - Send all messages after commit.
  - Break interactive transactions into smaller units of consistency that are transactions on their own.
- Another solution:
  - Define *compensating transactions*; this is not always possible.

Summary

- Undo/Redo
  - Before and after values written to log.
  - Writes occur whenever is convenient.
- No-Undo/Redo
  - Disk is written after commit.
  - No before values appear in log.
- Undo/No-Redo
  - Disk is written before commit.
  - No after values appear in log.
- No-Undo/Redo
  - Uses shadow paging.
  - Disk writes like undo/redo, except flush before commit record.
- Checkpoint
  - Flush buffers before writing checkpoint record