# Course Name:
# Database Management Systems

# Lecture 25
# Topics to be covered

- ❑ Recovery System
  - Failure Classification
  - Storage Structure
  - Recovery and Atomicity
  - Log-Based Recovery
  - Shadow Paging
  - Recovery With Concurrent Transactions
  - Buffer Management
  - Failure with Loss of Nonvolatile Storage
  - Advanced Recovery Techniques
  - ARIES Recovery Algorithm
  - Remote Backup Systems

# 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

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
  - Focus of this chapter
- 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 (battery backed up) RAM
- **Stable storage**:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media

# Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

# Stable-Storage Implementation (Cont.)

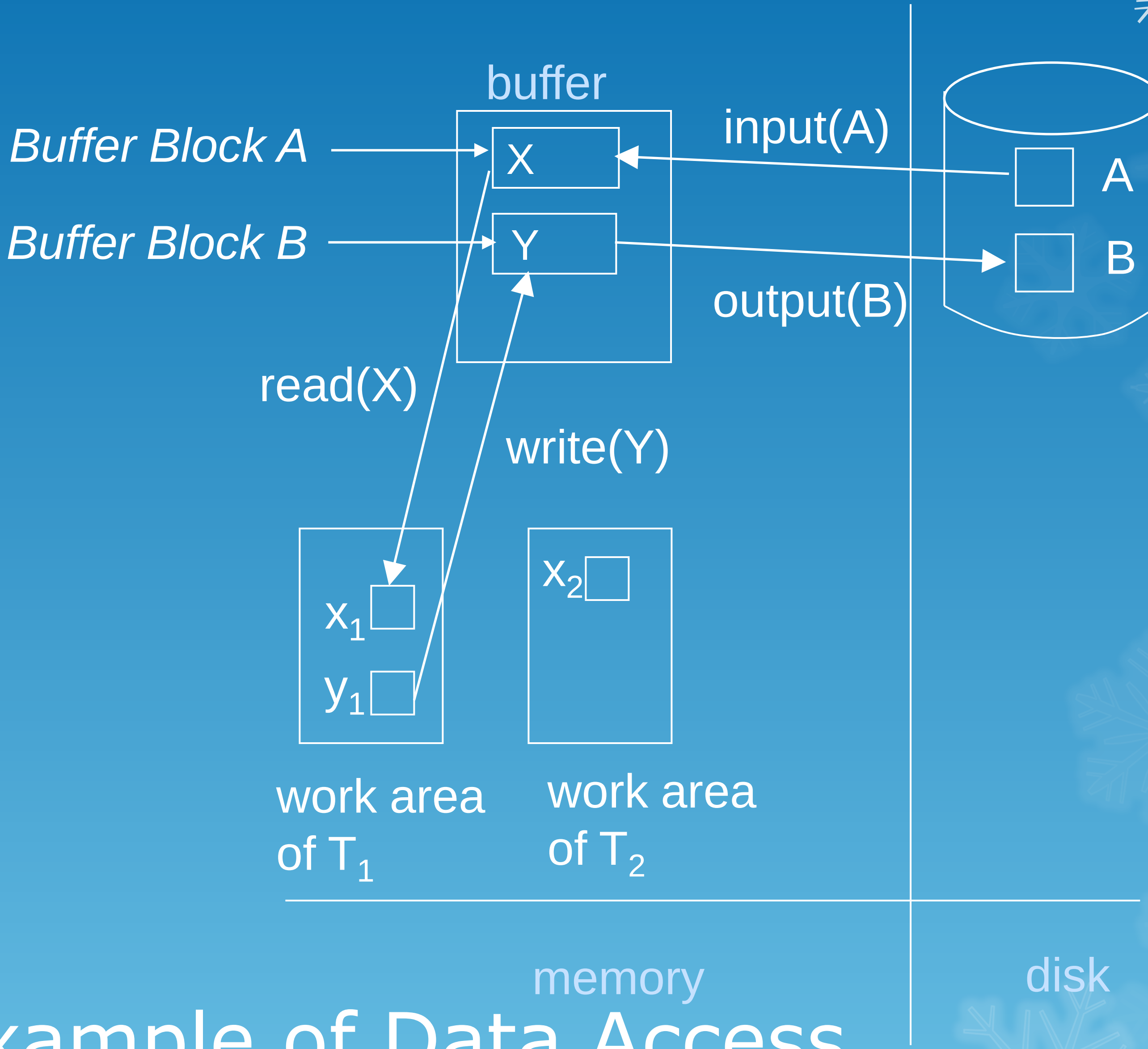
- Protecting storage media from failure during data transfer (cont.):
- Copies of a block may differ due to failure during output operation. To recover from failure:

1. First find inconsistent blocks:
    1. *Expensive solution*: Compare the two copies of every disk block.
    2. *Better solution*:
        - Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
        - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
        - Used in hardware RAID systems
2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

# 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.
- Each transaction $T_i$ has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - $T_i$'s local copy of a data item $X$ is called $x_i$.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

# Example of Data Access



# Recovery and Atomicity

- Modifying the database without ensuring that the transaction will commit  may leave the database in an inconsistent state.
- Consider transaction $T_i$ that transfers $50 from account *A* to account *B*;  goal is either to perform all database modifications made by $T_i$ or none at all.
- Several output operations may be required for $T_i$  (to output *A* and *B*). A failure may occur after one of these modifications have been made but before all of them are made.

# Recovery and Atomicity (Cont.)

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study two approaches:
  - **log-based recovery**, and
  - **shadow-paging**
- We assume (initially) that transactions run serially, that is, one after the other.

# 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
  <$T_i$ **start**>log record
- *Before* $T_i$ executes **write**($X$), a log record <$T_i$, $X$, $V_1$, $V_2$> is written, where $V_1$ is the value of $X$ before the write, and $V_2$ is the value to be written to $X$.
  - Log record notes that $T_i$ has performed a write on data item $X_j$. $X_j$ had value $V_1$ before the write, and will have value $V_2$ after the write.
- When $T_i$ finishes it last statement, the log record <$T_i$ **commit**> is written.
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

# Checkpoints

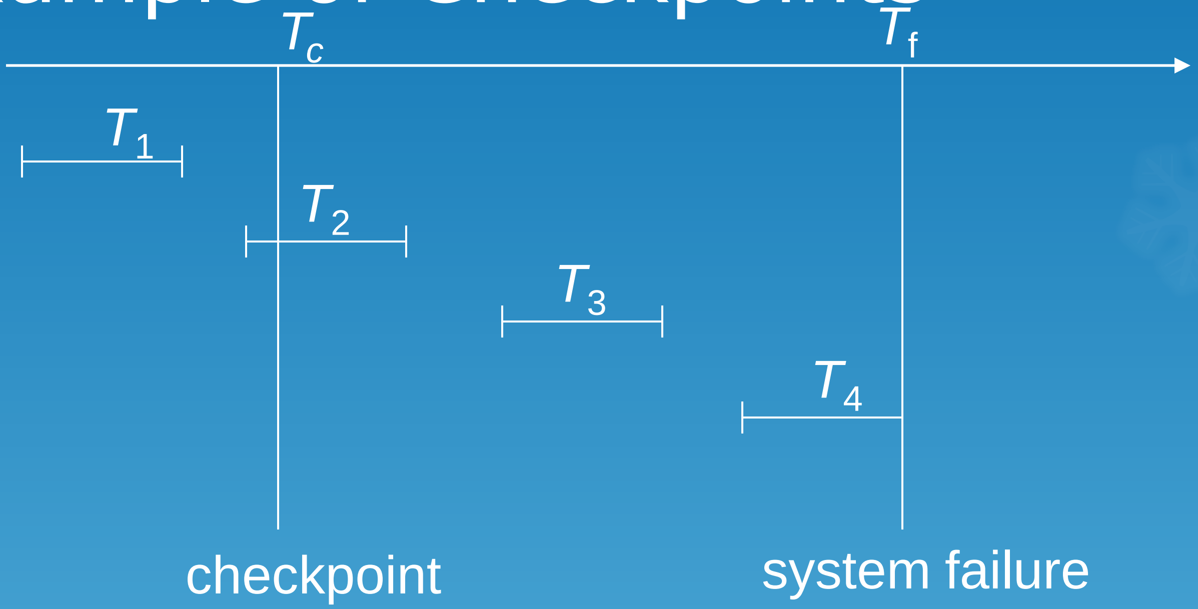
- Problems in recovery procedure
- searching the entire log is time-consuming
  1. we might unnecessarily redo transactions which have already
  2. output their updates to the database.
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently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

# 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**> record
  2. Continue scanning backwards till a record <$T_i$ **start**> is found.
  3. Need only consider the part of log following above **start** record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
  4. For all transactions (starting from $T_i$ or later) with no <$T_i$ **commit**>, execute **undo(**$T_i$*).* (Done only in case of immediate modification.)
  5. Scanning forward in the log, for all transactions starting from $T_i$ or later with a <$T_i$ **commit**>, execute **redo(**$T_i$*).*

# 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