

# Database Technology Recovery

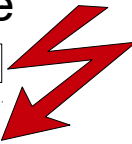


# Flashback to First Lecture

- We already stumbled upon transactions

```
Delete from file: active lecturers
Add to file: retired lecturers
```

Computer crashes here



File: active lecturers

```
Prof. Smith
Dr. Stevens
Prof. Miller
```

File: retired lecturers

```
Dr. Hawkins
Prof. Brown
Prof. Wilson
```

# Recap: ACID Properties

- **Atomicity:** Either all operations of the transaction are properly reflected in the database, or none
- **Consistency:** Execution of a full transaction preserves the consistency of the database
- **Isolation:** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions
  - Intermediate transaction results must be hidden from other concurrently executed transactions
  - i.e., for every pair of transactions  $T_i$  and  $T_j$ , it appears to  $T_i$  that either  $T_j$  finished execution before  $T_i$  started, or  $T_j$  started execution after  $T_i$  finished
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

# Outline

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- 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 as result of a 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

- Consider a transaction that transfers \$50 from account A to account B
  - Two updates: subtract 50 from A and add 50 to B
- Transaction 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
  - not 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
  - Actions taken *during normal transaction processing* to ensure enough information exists to recover from failures
  - 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
  - but may still fail, losing data
- **Stable storage:**
  - a mythical form of storage that survives all failures
  - approximation: maintaining multiple copies on distinct nonvolatile media

# Stable Storage Approximation

- 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 Approximation

- 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:
  - First find inconsistent blocks
    - Expensive solution: Compare the two copies of every disk block.
    - 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
  - 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

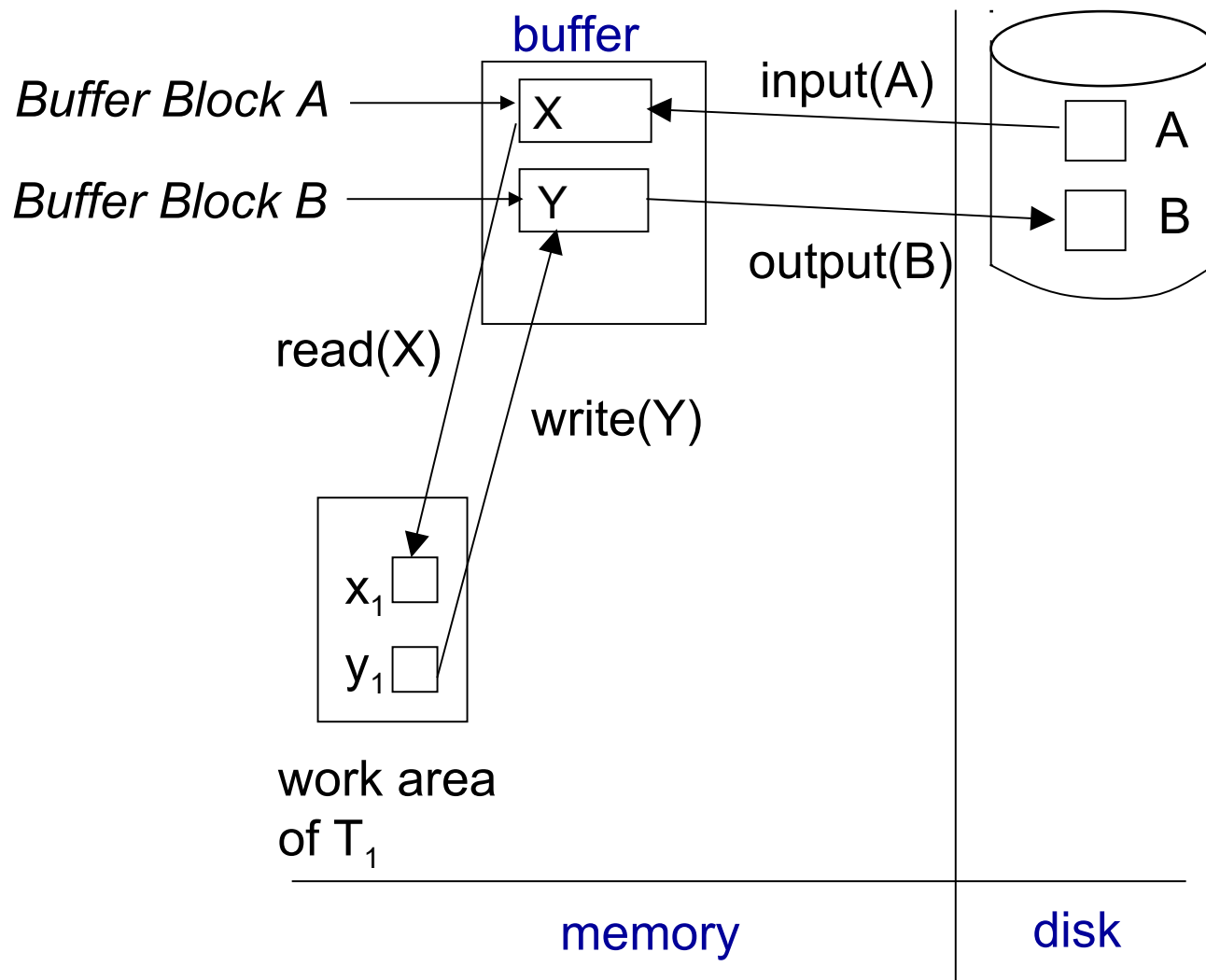
# Data Access and Buffering

- **Physical blocks** are those blocks residing on the disk
- **System 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
- Simplifying assumption:
  - each data item fits in, and is stored inside, a single block

# Data Access and Buffering

- 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 denoted by  $x_i$ .
  - $B_X$  denotes block containing  $X$
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read**( $X$ ) assigns the value of data item  $X$  to the local variable  $x_i$
  - **write**( $X$ ) assigns the value of local variable  $x_i$  to data item  $\{X\}$  in the buffer block
- Transactions
  - Must perform **input**( $X$ ) before accessing  $X$  for the first time (subsequent reads can be from local copy)
  - The **write**( $X$ ) can be executed at any time before the transaction commits
- Note that **output**( $B_X$ ) need not immediately follow **write**( $X$ )
  - system can perform the **output** operation when it deems fit

# Data Access and Buffering



# Recovery and Atomicity

- How can we ensure atomicity despite failures?
  - *first* output information describing the modification to stable storage
  - *then* modify the database itself
- This is called a **log-based recovery mechanism**
- For the moment, we assume serial execution for simplicity
  - parallel variants exist

# Log-based Recovery

- A **log** is kept on stable storage
  - sequence of **log records**
  - maintains information about update activities on the database
- When transaction  $T_i$  starts, it registers itself by writing a record  $\langle T_i \text{ start} \rangle$  to the log
- Before  $T_i$  executes **write**( $X$ ), a log record is written  $\langle T_i, X, V_1, V_2 \rangle$  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

# 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** a database item is written
    - we assume that the log record is output immediately to stable storage
    - however, it is possible to postpone log record output to some extent
  - Output of updated blocks to disk 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
- For the moment, we only consider the immediate modification scheme

# 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
-----	-------	--------

**<T<sub>0</sub> start>**

**<T<sub>0</sub>, A, 1000, 950>**

**<T<sub>0</sub>, B, 2000, 2050>**

A = 950  
B = 2050

**<T<sub>0</sub> commit>**

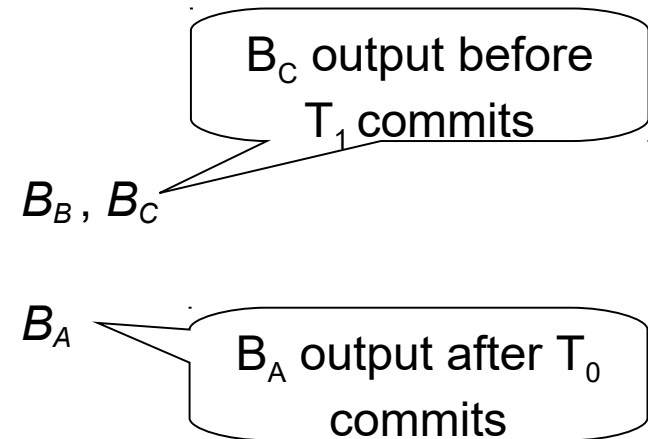
**<T<sub>1</sub> start>**

**<T<sub>1</sub>, C, 700, 600>**

C = 600

**<T<sub>1</sub> commit>**

- Note:  $B_X$  denotes block containing X



# 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 (called **redo-only**)  $\langle T_i, X, V \rangle$  is written out
    - When undo of a transaction is complete, a log record  $\langle T_i, \mathbf{abort} \rangle$  is written out (to indicate that the undo was completed)
  - **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 Operations

- **undo** and **redo** operations are used in several different circumstances:
- **undo is** used
  - for *transaction rollback* during normal operation (in case a transaction cannot complete its execution due to some logical error)
- **undo** and **redo** operations are used during recovery from failure
  - We need to deal with the case where during recovery from failure, another failure occurs prior to the system having fully recovered

# Transaction Rollback

- 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  $\langle T_i, \mathbf{start} \rangle$  is found stop the scan and write the log record  $\langle T_i, \mathbf{abort} \rangle$

# 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  $\langle T_i \text{ abort} \rangle$
- Why redo transaction  $T_i$  if the case of  $\langle T_i \text{ abort} \rangle$ ?
  - for  $\langle T_i \text{ abort} \rangle$ , there are also redo-only records for the undo operation
  - the end result will be to undo  $T_i$ 's modifications in this case
  - redo *all* original actions including the steps that restored the old value
    - known as *repeating history*
  - simplifies the recovery algorithm, enables faster overall recovery time

# Examples for Immediate Recovery

- 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
  - All updates are stopped while doing checkpointing
    - Output all log records currently residing in main memory onto stable storage
    - Output all modified buffer blocks to the disk
    - Write a log record <checkpoint L> onto stable storage where L is a list of all transactions active at the time of checkpoint

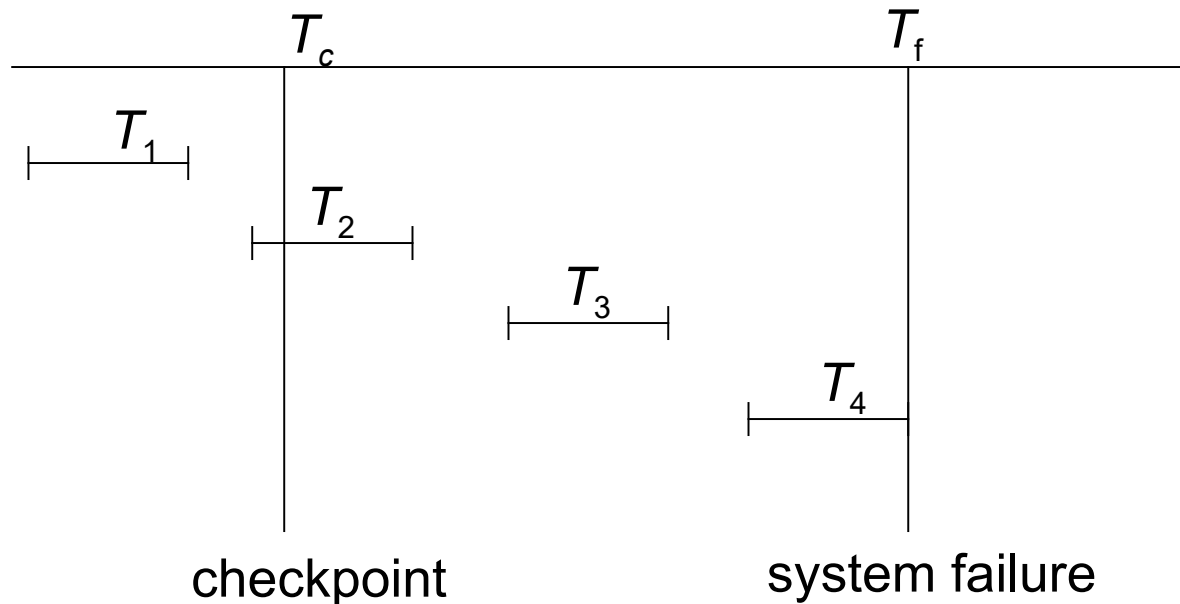
# Checkpoints

- 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



# Checkpoints

- $T_1$  can be ignored
  - updates have already been output to disk due to checkpoint
- $T_2$  and  $T_3$  are redone
- $T_4$  is undone



# The 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  $\langle T_i \text{ abort} \rangle$

# The Recovery Algorithm

- **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

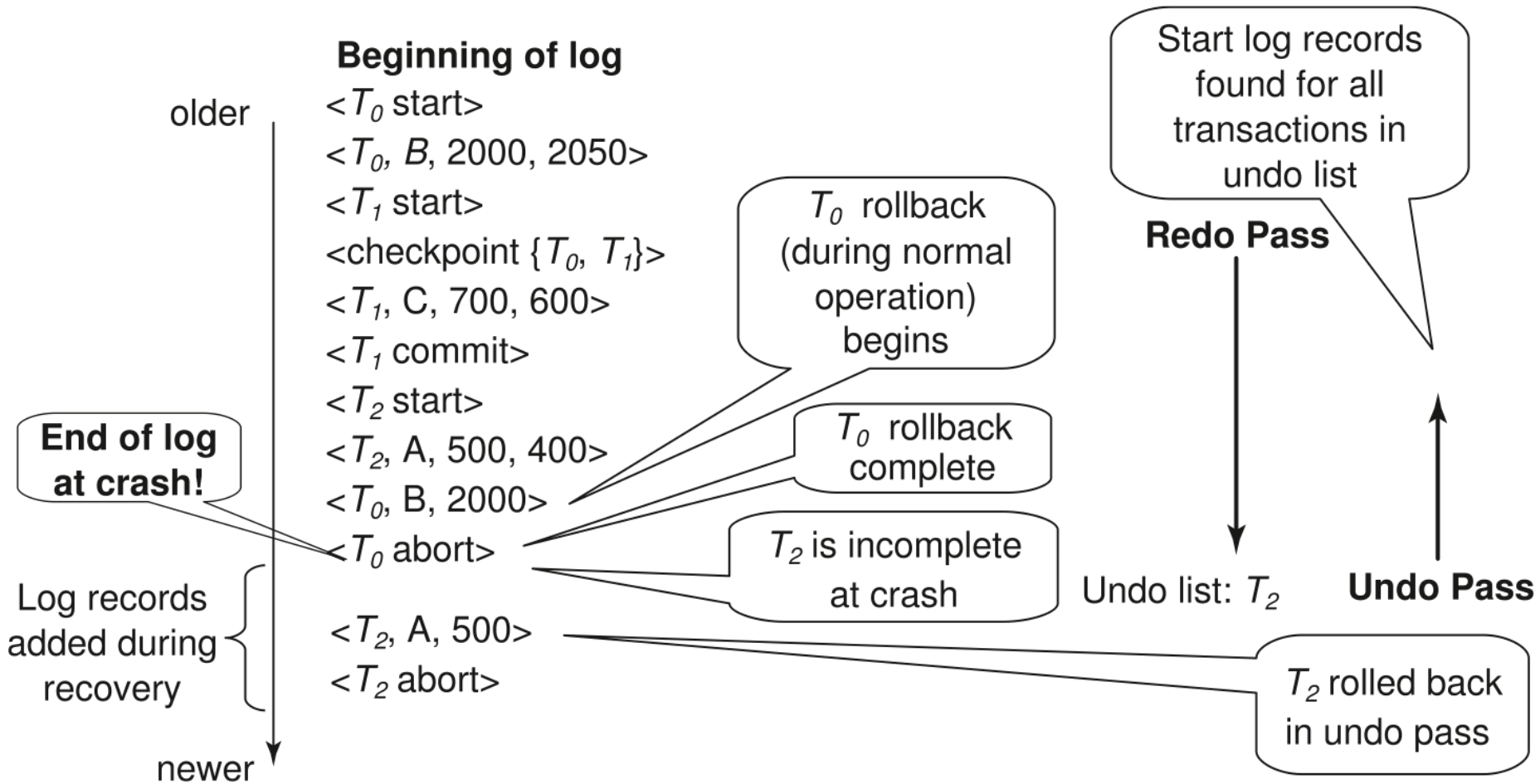
# The Recovery Algorithm

- **Redo phase:**
  - Find last **<checkpoint L>** record, and set undo-list to  $L$ .
  - Scan forward from above **<checkpoint L>** record
    - Whenever a record  $\langle T_i, X_j, V_1, V_2 \rangle$  is found,
      - redo it by writing  $V_2$  to  $X_j$
    - Whenever a log record **<T<sub>i</sub> start>** is found
      - add  $T_i$  to undo-list
    - Whenever a log record **<T<sub>i</sub> commit>** or **<T<sub>i</sub> abort>** is found
      - remove  $T_i$  from undo-list

# The Recovery Algorithm

- **Undo phase:**
  - Scan log backwards from end
  - When 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:
      - perform undo by writing  $V_1$  to  $X_j$ .
      - write a log record  $\langle T_i, X_j, V_1 \rangle$
  - When a log record  $\langle T_i \text{ start} \rangle$  is found where  $T_i$  is in undo-list
    - Write a log record  $\langle T_i \text{ abort} \rangle$
    - Remove  $T_i$  from undo-list
  - 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

# Recovery Example



# Log Record Buffering

- **Log record buffering:**
  - log records are buffered in main memory
  - instead of being output directly to stable storage
- Log records are output to stable storage
  - when a block of log records in the buffer is full
  - or a **log force** operation is executed
- Log force is performed to commit a transaction
  - by forcing all its log records (including the commit record) to stable storage
  - Several log records can thus be output using a single output operation, reducing the I/O cost

# Log Record Buffering & Write-Ahead Logging

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order of creation
  - Transaction  $T_i$  enters the commit state only when the log record  $\langle T_i \text{ commit} \rangle$  has been output to stable storage
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage
    - This rule is called the **write-ahead logging** or **WAL** rule

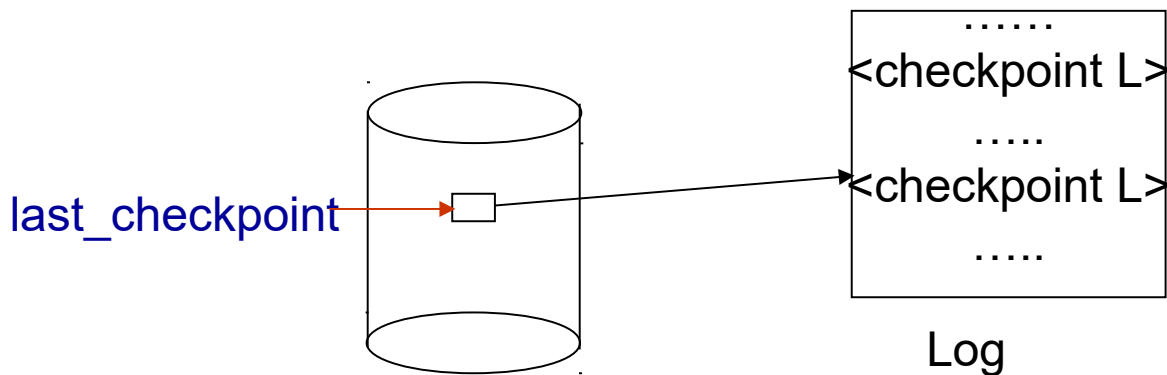


# Fuzzy Checkpointing

- To avoid long interruption of normal processing during checkpointing, allow updates to happen during checkpointing
- Fuzzy checkpointing is done as follows:
  - Temporarily stop all updates by transactions
  - Write a <checkpoint L> log record and force log to stable storage
  - Note list M of modified buffer blocks
  - Now permit transactions to proceed with their actions
  - Output to disk all modified buffer blocks in list M
    - blocks should not be updated while being output
    - Follow WAL: all log records pertaining to a block must be output before the block is output
  - Store a pointer to the checkpoint record in a fixed position last\_checkpoint on disk

# Fuzzy Checkpointing

- When recovering using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last\_checkpoint**
  - Log records before **last\_checkpoint** have their updates reflected in database on disk, and need not be redone
  - Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely



# Failure with Loss of Non-volatile Storage

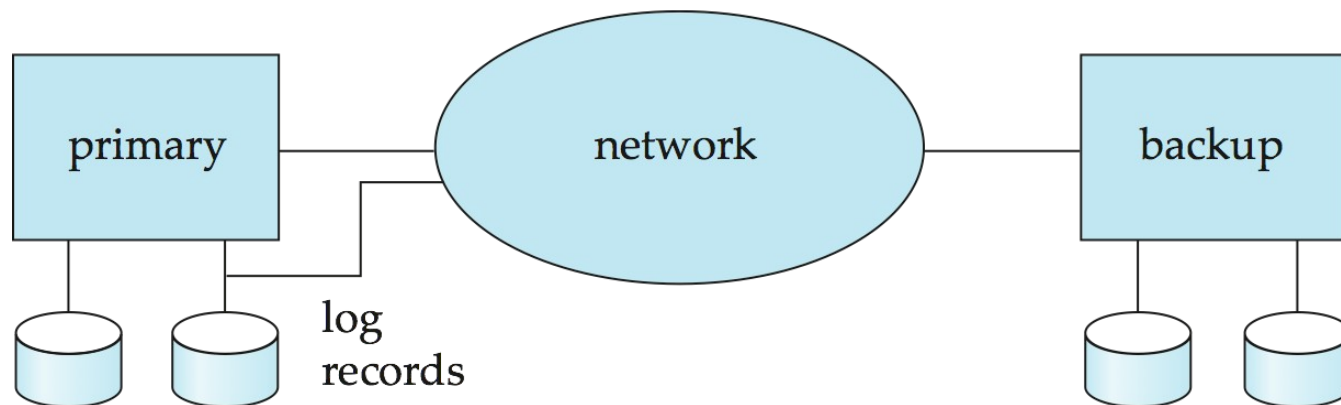
- So far we assumed no loss of non-volatile storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure
  - a procedure similar to checkpointing must take place
- Output all log records currently in main memory onto stable storage
  - Output all buffer blocks onto the disk
  - Copy the contents of the database to stable storage
  - Output a record **<dump>** to log on stable storage

# Recovery from Failure of Non-volatile Storage

- To recover from disk failure
  - restore database from most recent dump
  - consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump; known as **fuzzy dump** or **online dump**
  - Similar to fuzzy checkpointing

# Remote Backup Systems

- Risk minimization:
  - allowing transaction processing to continue even if the primary site is destroyed



# Remote Backup Systems

- **Detection of failure:**
  - Backup site must detect when primary site has failed
  - to distinguish primary site failure from link failure: maintain *several* communication links in between
  - Heart-beat messages
- **Transfer of control:**
  - To take over control backup site first performs recovery using its copy of the database and all the log records it has received from the primary
    - i.e., completed transactions are redone and incomplete transactions are rolled back
  - When the backup site takes over processing it becomes the new primary
  - To transfer control back to old primary when it recovers
    - old primary must receive redo logs from the old backup and apply all updates locally

# Remote Backup Systems

- **Time to recover** – reduce delay in takeover
  - backup site periodically processes the redo log records
    - i.e., performs recovery from previous database state
  - performs a checkpoint, and can then delete earlier parts of the log
- Hot-Spare configuration permits very fast takeover:
  - Backup continually processes redo log records as they arrive
    - applying the updates locally
  - When failure of the primary is detected
    - the backup rolls back incomplete transactions
    - and is ready to process new transactions
- Alternative to remote backup: distributed database with replicated data
  - Remote backup is faster and cheaper, but less tolerant to failure

# Remote Backup Systems

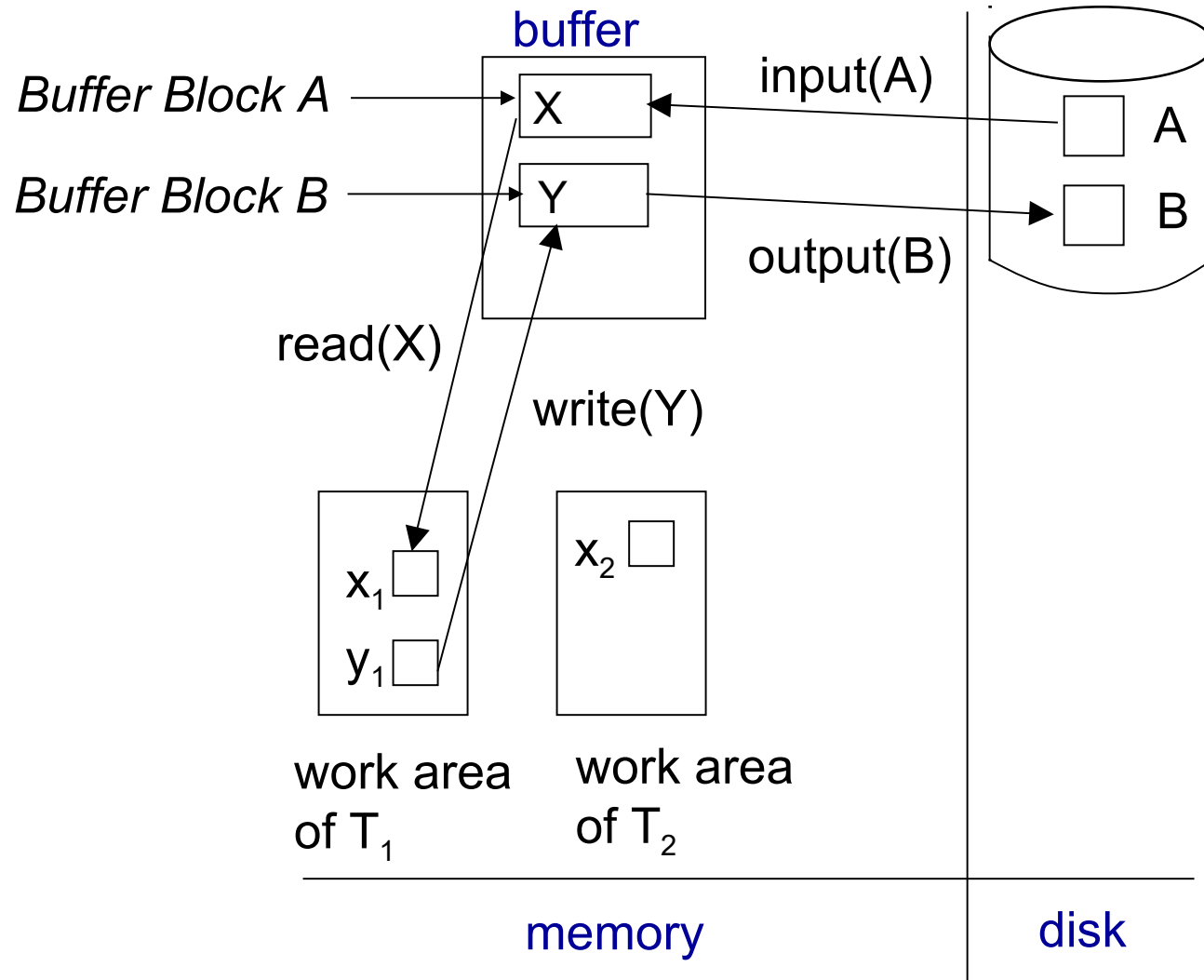
- Ensure durability of updates by delaying transaction commit
  - until update is logged at backup
  - avoid this delay by permitting lower degrees of durability
- One-safe: commit as soon as transaction's commit log record is written at primary
  - Problem: updates may not arrive at backup before it takes over
- Two-very-safe: commit when transaction's commit log record is written at primary and backup
  - Reduces availability since transactions cannot commit if either site fails
- Two-safe: proceed as in two-very-safe if both primary and backup are active
  - If only the primary is active, the transaction commits as soon as its commit log record is written at the primary
  - Better availability than two-very-safe
  - avoids problem of lost transactions in one-safe



# Concurrency Control and Recovery

- All transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions
- Log records of different transactions may be interspersed in the log
- 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 do we 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)

# Data Access and Buffering (revisited)



# Summary

- Recovery ensures consistency of the database
  - handles rollbacks
  - takes care of setting the database back to operation after failures
- Mechanisms
  - Logs: write ahead (write log first, then write data)
  - Checkpoints
- Trade off between normal and recovery performance
  - e.g., by using fuzzy checkpoints
- Remote backup
  - distribution of risk
- Recovery and Concurrency

# Questions?

