Chapter 16: Recovery System

Database System Concepts, 6th Ed.

Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Buffer Management
- Fuzzy Checkpointing
- 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 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

Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Buffer Management
- Fuzzy Checkpointing
- Remote Backup Systems

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

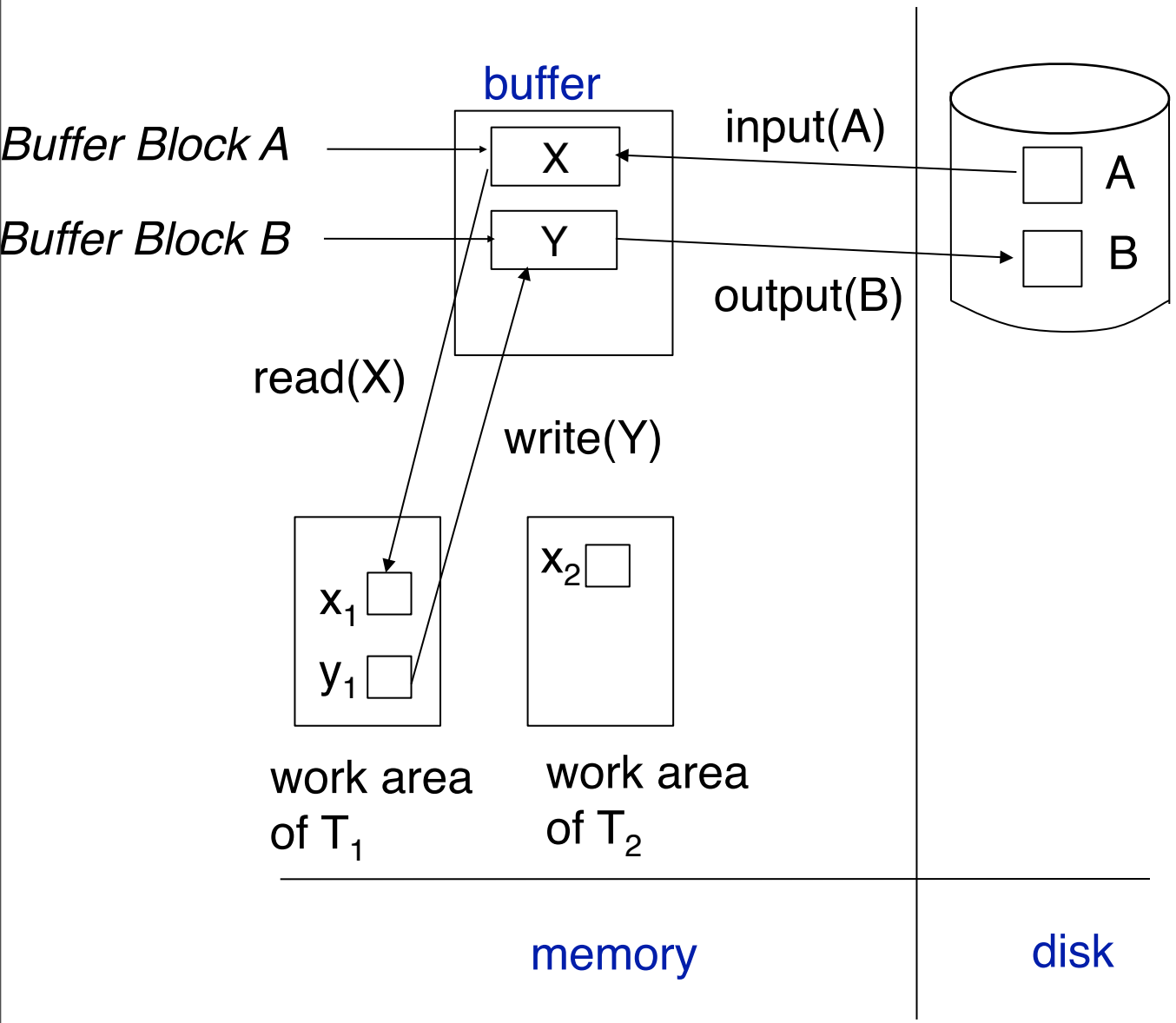
Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Protecting storage media from failure during data transfer (one solution that assumes 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.)

- To recover from failure (copies of a block may differ due to failure during output operation):
  1. First find inconsistent blocks:
     1. *Expensive solution*: Compare the two copies of every disk block.
     2. *Better solution (logging)*:
        - 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:
  - **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.

Data Access (Cont.)

- 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.
- 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.
  - **Note: output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.
- Transactions
  - Must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
  - **write**(X) can be executed at any time before the transaction commits

Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Buffer Management
- Fuzzy Checkpointing
- Remote Backup Systems

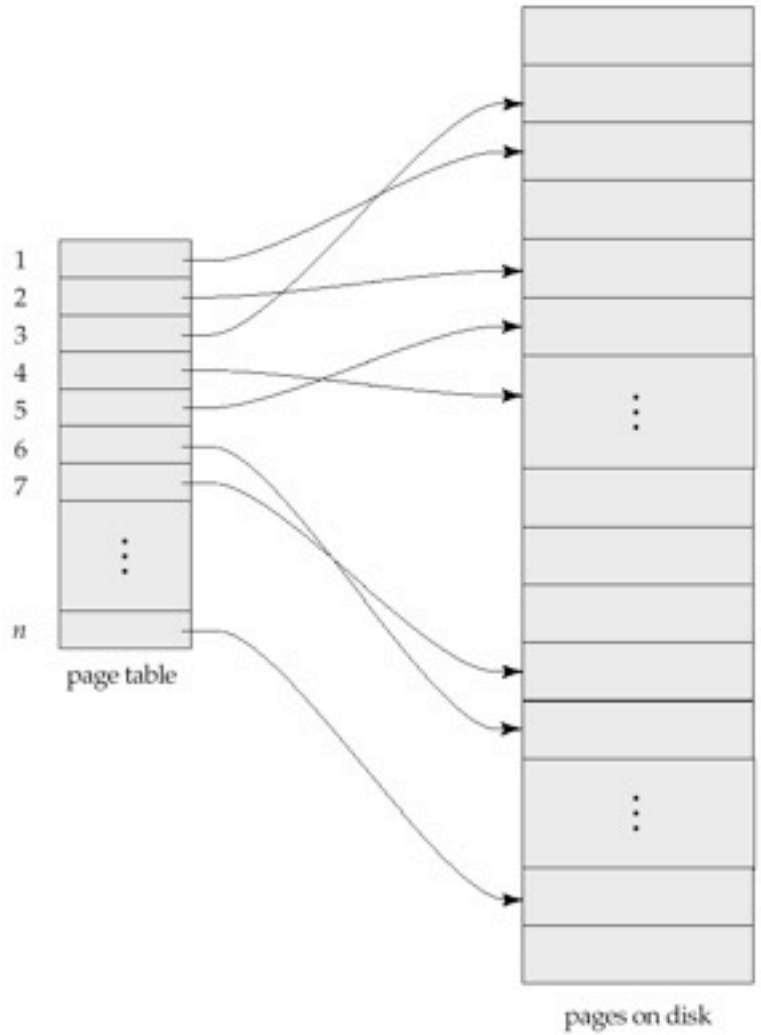
Recovery and Atomicity

- 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** in detail
  - We first present key concepts
  - And then present the actual recovery algorithm
- Less used alternative: **shadow-paging** (brief details in book)

Shadow Paging

- **Shadow paging** is an alternative to log-based recovery.
- Idea: maintain *two* page tables – 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
- Both the page tables are initially 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

Sample Page Table



Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Buffer Management
- Fuzzy Checkpointing
- Remote Backup Systems

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 (the **old value**), and V_2 is the value to be written to X (the **new value**).
- When T_i finishes it last statement, the log record $<T_i$ **commit**> is written (indicating that T_i is committed).

Transaction Commit

- A transaction is said to have been committed when its commit log record is output to stable storage
  - all previous log records of the transaction must have been output already (*may have been kept in memory for performance*)
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

Database Modification Example

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_0, B, 2000, 2050>		
	$A = 950$	
	$B = 2050$	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	$C = 600$	
		B_B, B_C
<T_1 **commit**>		
		B_A

B_C output before T_1 commits

B_A, B_B output after T_0 commits

- Note: B_X denotes block containing X.

Undo and Redo Operations

- **Undo** of a log record $<T_i, X, V_1, V_2>$ writes the **old** value V_1 to X
- **Redo** of a log record $<T_i, X, V_1, V_2>$ 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 $<T_i, X, V>$ is written out
    - when undo of a transaction is complete, a log record $<T_i$ **abort**$>$ 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*

DB Modification Recovery Example

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

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

DB Modification Recovery Example

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

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) **undo (T_0)**: B is restored to 2000 and A to 1000, and log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> are written out

DB Modification Recovery Example

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

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) **undo (T_0)**: B is restored to 2000 and A to 1000, and log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> 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 <T_1, C, 700>, <T_1, **abort**> are written out.

DB Modification Recovery Example

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

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) **undo (T_0)**: B is restored to 2000 and A to 1000, and log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> 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 <T_1, C, 700>, <T_1, **abort**> 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

Undo and Redo on Recovering from Failure

- When recovering after failure:
  - Transaction T_i needs to be undone if the log
    - contains the record $<T_i$ **start**$>$,
    - but does not contain either the record $<T_i$ **commit**$>$ *or* $<T_i$ **abort**$>$.
  - Transaction T_i needs to be redone if the log
    - contains the records $<T_i$ **start**$>$
    - and contains the record $<T_i$ **commit**$>$ *or* $<T_i$ **abort**$>$

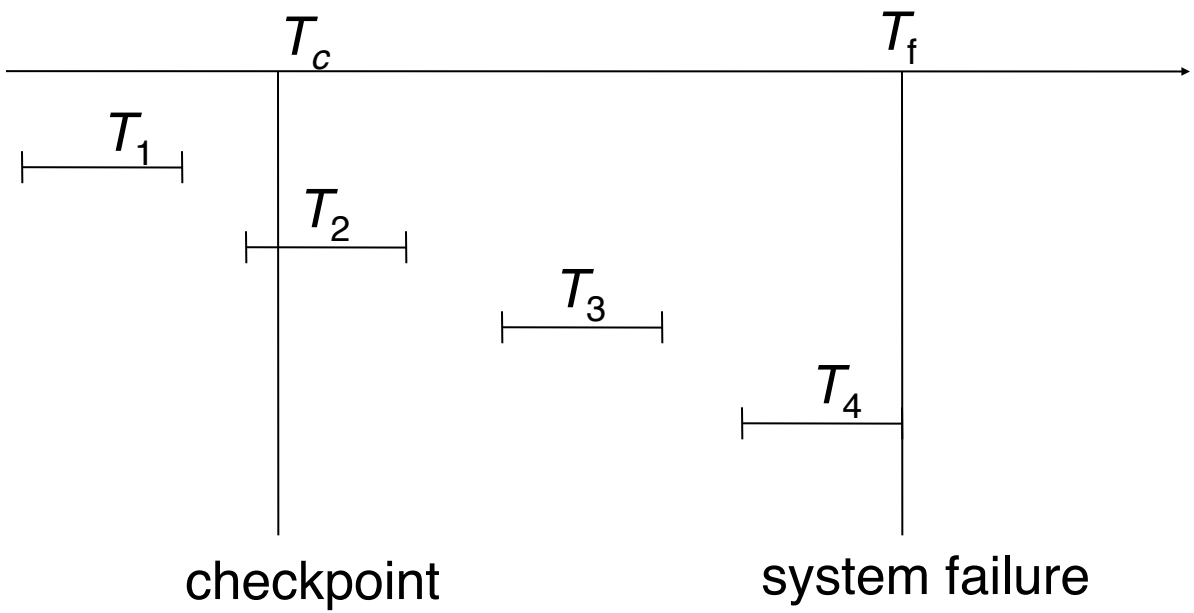
Checkpoints

- 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**
  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.

Checkpoints (Cont.)

- During recovery we need to do the following:
  - 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.

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

Example of Recovery

- Go over the steps of the recovery algorithm on the following log:

  <T_0 **start**>

  <T_0, A, 0, 10>

  <T_0 **commit**>

  <T_1 **start**> /* Scan at step 1 comes up to here */

  <T_1, B, 0, 10>

  <T_2 **start**>

  <T_2, C, 0, 10>

  <T_2, C, 10, 20>

  <checkpoint {T_1, T_2}>

  <T_3 **start**>

  <T_3, A, 10, 20>

  <T_3, D, 0, 10>

  <T_3 **commit**>

Example of Recovery

- Go over the steps of the recovery algorithm on the following log:

  <T_0 **start**>

  <T_0, *A*, 0, 10>

  <T_0 **commit**>

  <T_1 **start**> /* Scan at step 1 comes up to here */

  <T_1, *B*, 0, 10>

  <T_2 **start**>

  <T_2, *C*, 0, 10>

  <T_2, *C*, 10, 20>

  <checkpoint {T_1, T_2}>

  <T_3 **start**>

  <T_3, *A*, 10, 20>

  <T_3, *D*, 0, 10>

  <T_3 **commit**>

  redo: T_3
  undo: T_1, T_2

Recovery Algorithm

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

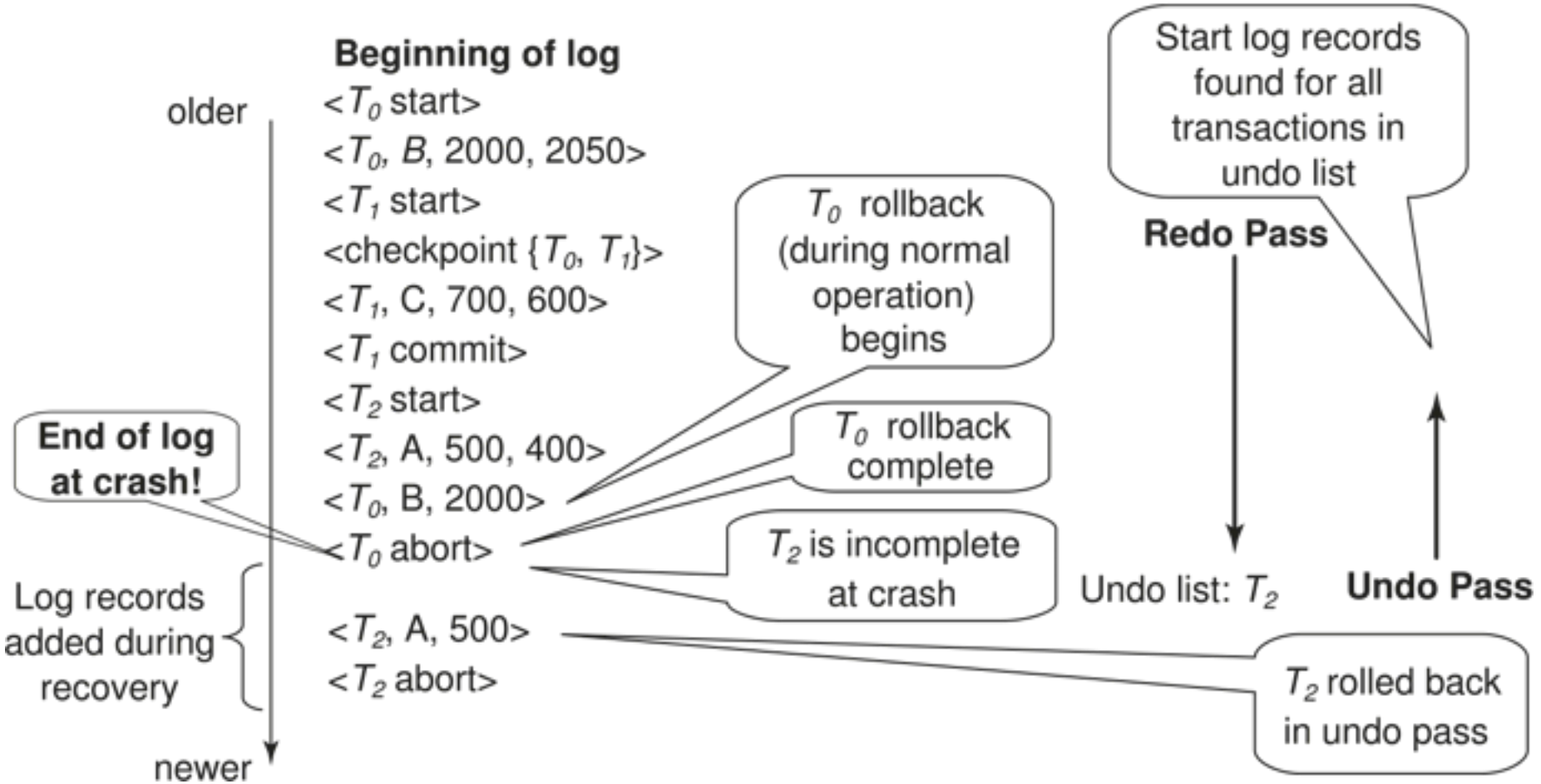
Recovery Algorithm (Cont.)

- **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 the above <**checkpoint** *L*> record
     1. Whenever a record $<T_i, X_j, V_1, V_2>$ is found, redo it by writing V_2 to X_j
     2. Whenever a log record $<T_i$ **start**> is found, add T_i to undo-list
     3. Whenever a log record $<T_i$ **commit**> *or* $<T_i$ **abort**> is found, remove T_i from undo-list

Recovery Algorithm (Cont.)

- **Undo phase:**
  1. Scan log backwards from end
     1. Whenever a log record $<T_i, X_j, V_1, V_2>$ 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 $<T_i, X_j, V_1>$
     2. Whenever a log record $<T_i$ **start**$>$ is found where T_i is in undo-list,
        1. Write a log record $<T_i$ **abort**$>$
        2. Remove T_i from undo-list
     3. Stop when undo-list is empty
        - i.e. $<T_i$ **start**$>$ has been found for every transaction in undo-list
- After undo phase completes, normal transaction processing can commence

Example of Recovery



Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- **Buffer Management**
- Fuzzy Checkpointing
- Remote Backup Systems

Log Record Buffering

- **Log record buffering**: log records are buffered in main memory, instead of 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.

Database Buffering

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  - If the block chosen for removal has been updated, it must be output to disk
- The recovery algorithm supports the **no-force policy** (i.e., updated blocks need not be written to disk when transaction commits)
  - **force policy**: requires updated blocks to be written at commit (more expensive)
- The recovery algorithm supports the **steal policy** (i.e., blocks containing updates of uncommitted transactions can be written to disk, even before the transaction commits)

Database Buffering (Cont.)

- If a block with uncommitted updates is output to disk, log records for the updates are output to the log on stable storage first
  - (Write ahead logging)
- No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
  - Before writing a data item, transaction acquires exclusive lock on block containing the data item
  - Lock can be released once the write is completed.
    - Such locks held for short duration are called **latches**.
- **To output a block to disk**
  1. First acquire an exclusive latch on the block (ensures no update can be in progress on the block)
  2. Then perform a **log flush**
  3. Then output the block to disk
  4. Finally release the latch on the block

Chapter 16: Recovery System

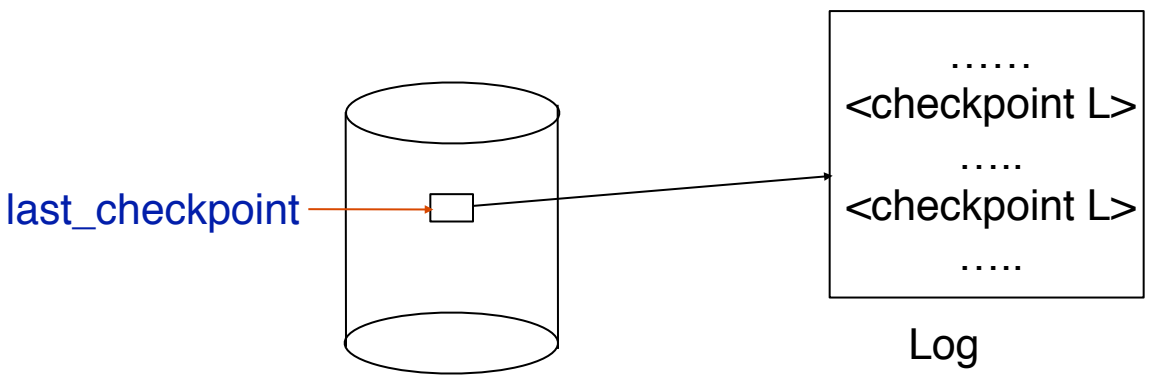
- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Buffer Management
- Fuzzy Checkpointing
- Remote Backup Systems

Fuzzy Checkpointing

- To avoid long interruption of normal processing during checkpointing, allow updates to happen during checkpointing
- **Fuzzy checkpointing** is done as follows:
  1. Temporarily stop all updates by transactions
  2. Write a <**checkpoint** *L*> log record and force log to stable storage
  3. Note list *M* of modified buffer blocks
  4. Now permit transactions to proceed with their actions
  5. 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
  6. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk

Fuzzy Checkpointing (Cont.)

- 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

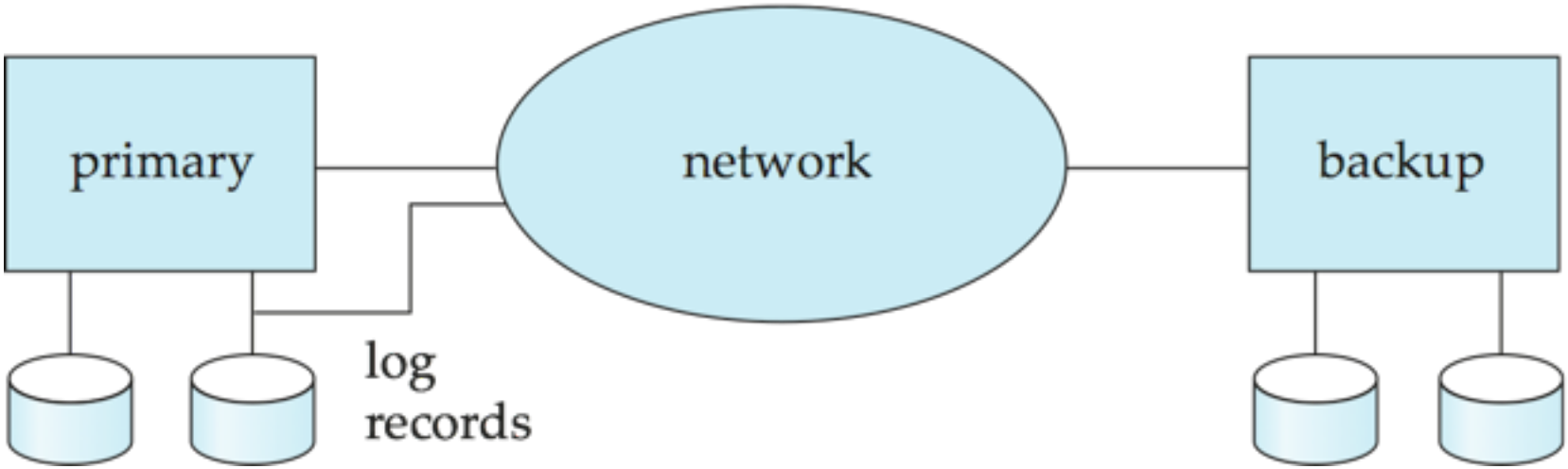


Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Buffer Management
- Fuzzy Checkpointing
- Remote Backup Systems

Remote Backup Systems

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.



Remote Backup Systems (Cont.)

- **Detection of failure**: Backup site must detect when primary site has failed
  - to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.
  - Heart-beat messages
- **Transfer of control**:
  - To take over control, the backup site first performs recovery using its copy of the database and all the log records it has received from the primary.
    - Thus, 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 (Cont.)

- **Time to recover**: To reduce delay in takeover, backup site periodically processes the redo log records (in effect, performing 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 record 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.

Remote Backup Systems (Cont.)

- 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 is commit log record is written at the primary.
  - Better availability than two-very-safe; avoids problem of lost transactions in one-safe.

Chapter 16: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Buffer Management
- Fuzzy Checkpointing
- Remote Backup Systems