Chapter 16: Recovery System

Database System Concepts, 6th Ed.

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

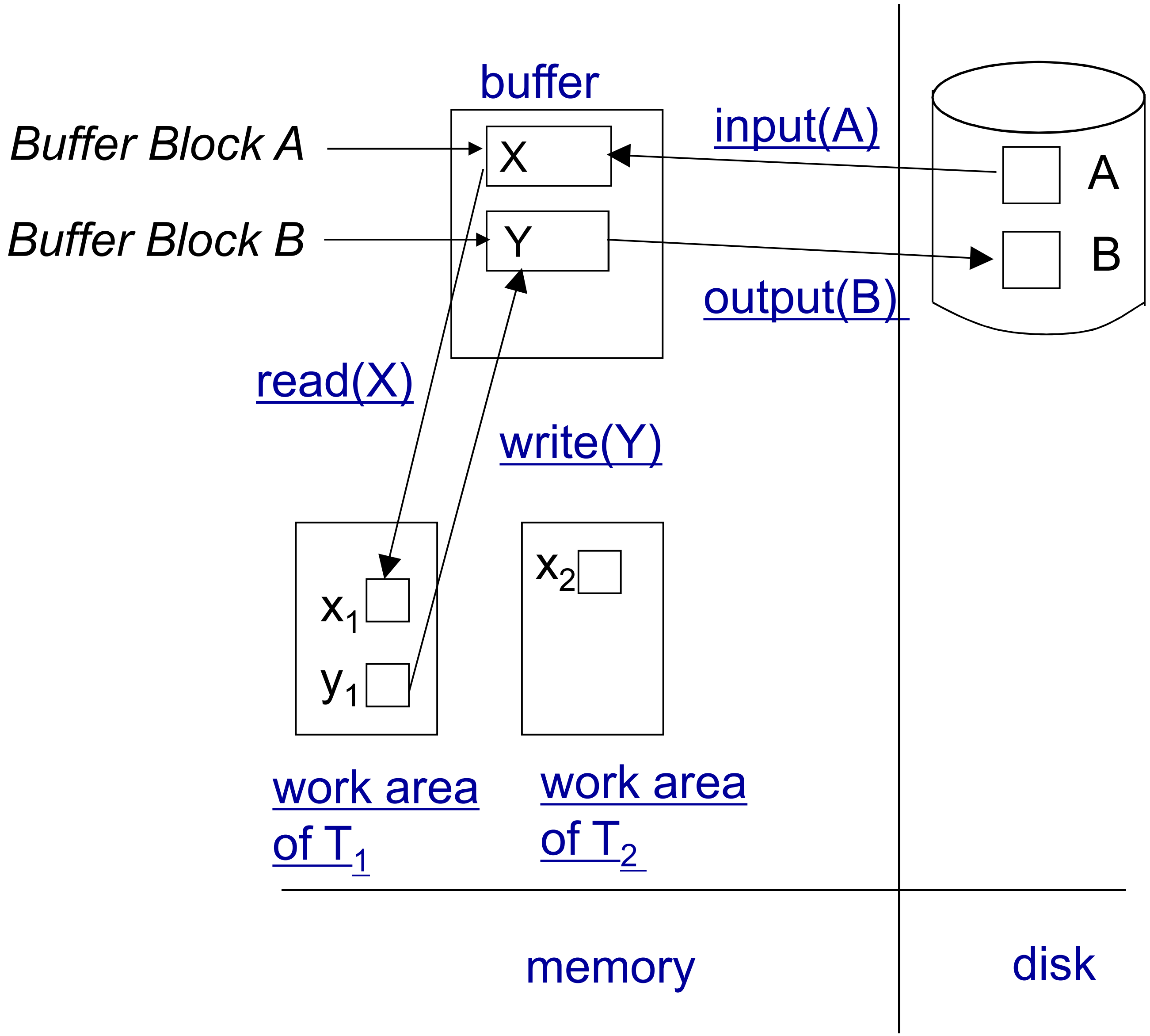
- Consider transaction T_i that transfers \$50 from account *A* to account *B*
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database
  - A failure may occur after one of these modifications have been made but before both of them are made
  - Modifying the database without 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
  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

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
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block

Example of Data Access



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

Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself
  - **Stable storage**: a mythical form of storage that survives all failures
    - approximated by maintaining multiple copies on distinct nonvolatile media
- We study **log-based recovery mechanisms** in detail
  - We first present key concepts
  - And then present the actual recovery algorithm

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

- 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/Deferred Database Modification

Two approaches using logs

- **Immediate-modification** scheme: allows updates of an uncommitted tx to be made to the buffer, or the disk itself, before the tx commits
  - Update log record must be written *before* database item is written
    - We assume that the log record is output directly to stable storage
  - Output of updated blocks to stable storage can take place at any time before or after tx commit
  - Order in which blocks are output can be different from the order in which they are written
- **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

Immediate Database Modification Example

- Example transactions T_0 and T_1 (T_0 executes before T_1):

  T_0: **read** (A)
  $C = C - 100$
  Write (A)
  read (B)
  $B = B + 50$
  write (B)

  T_1 : **read** (C)
  $A = A – 50$
  write (C)

Log	**Write**	**Output**
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_o, 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 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

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**>

- Note that If transaction T_i was undone earlier and the <T_i **abort**> record written to the log, and then a failure occurs, on recovery from failure T_i is redone
  - Such a redo redoes all the original actions ***including the steps that restored old values***
    - Known as **repeating history**
    - Seems wasteful, but simplifies recovery greatly

Immediate 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:

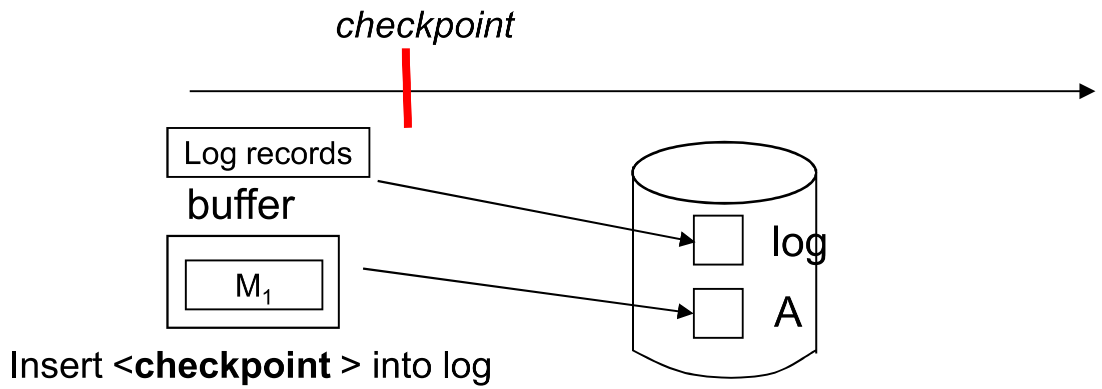
(a) undo (T_0): B is restored to 2000 and A to 1000, and log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> are written out.

(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.

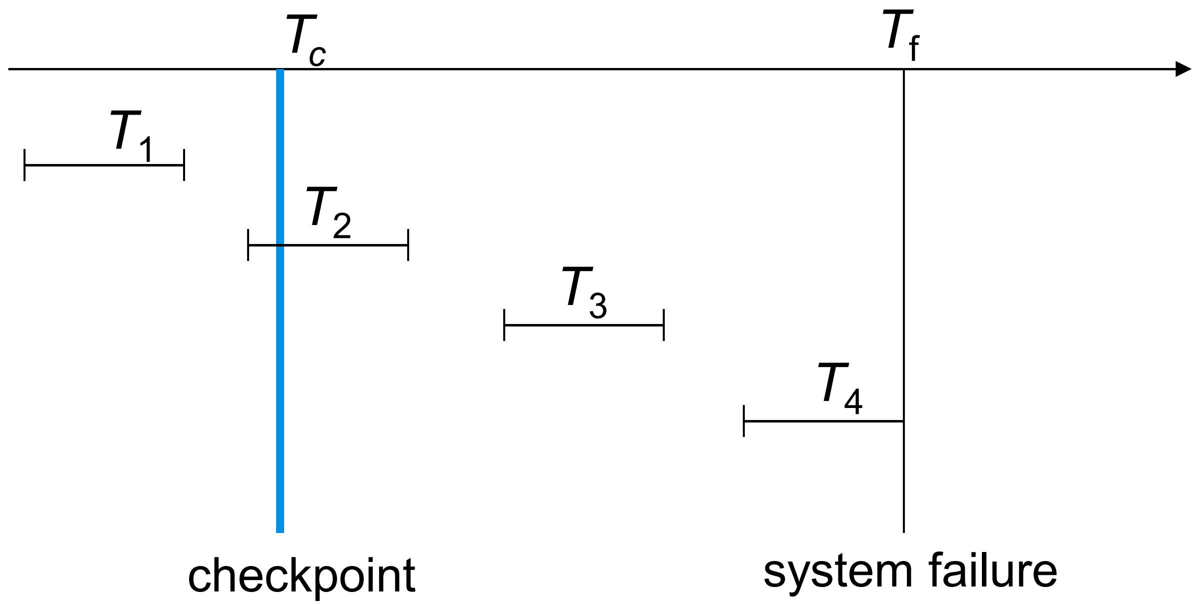
(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600.

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
  - All updates are stopped while doing checkpointing



Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- *Undo* T_4
- *Redo* T_2 and T_3

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)**
  - 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 $<T_i, X_j, V_1, V_2>$
    - 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 – Redo Phase

- **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

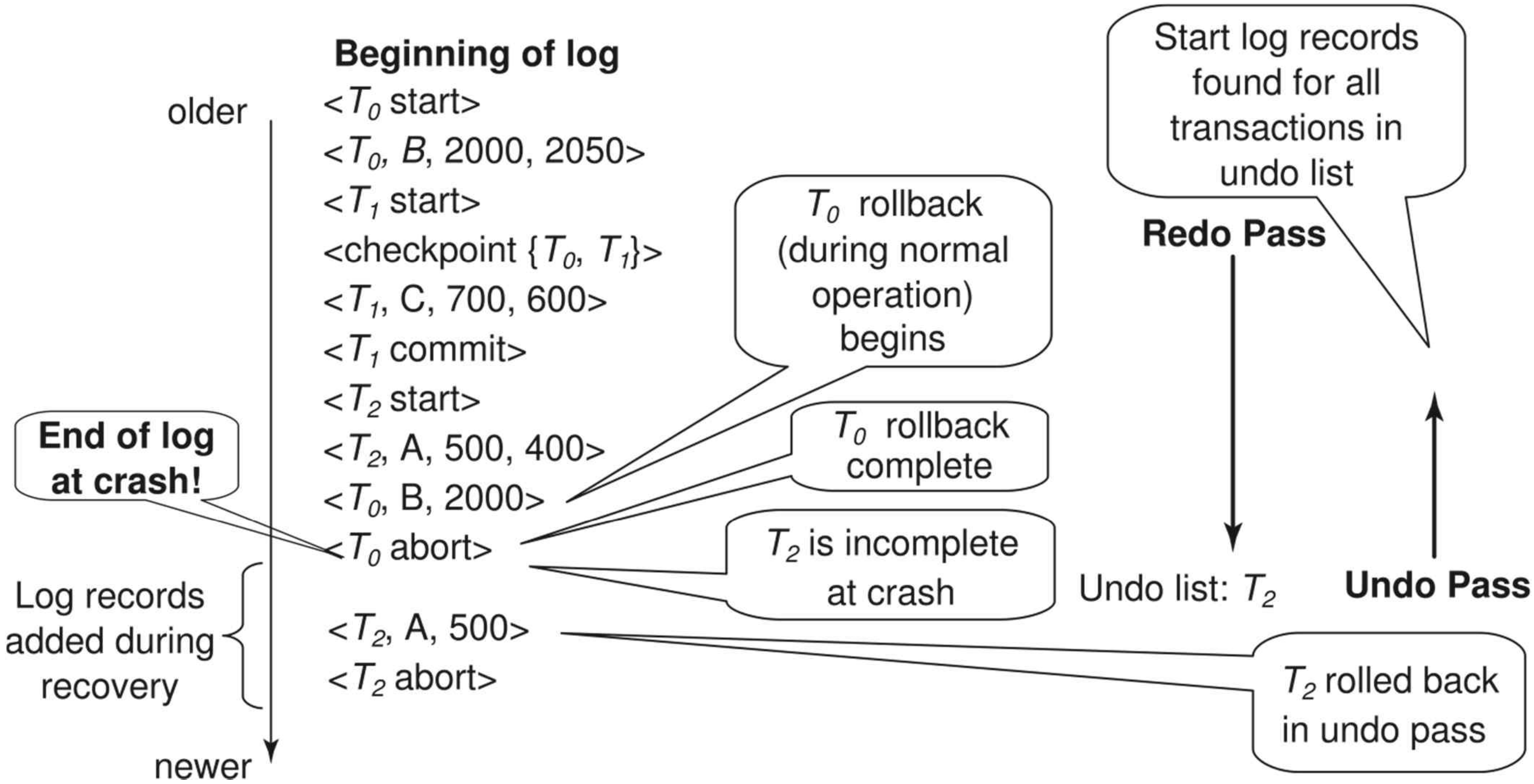
- **Redo phase**:
  1. Find last <**checkpoint** *L*> record, and set undo-list to *L*.
  2. Scan forward from above <**checkpoint** *L*> record
     1. Whenever a record $<T_i, X_j, V_1, V_2>$ is found, redo it by writing V_2 to X_j
     2. Whenever a log record $<T_i$ **start**$>$ is found, add T_i 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 – Undo Phase

- **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



End of Chapter 16

Database System Concepts, 6th Ed.