Database Development and Design (CPT201)

Lecture 6c: Transaction Management – Failure Recovery

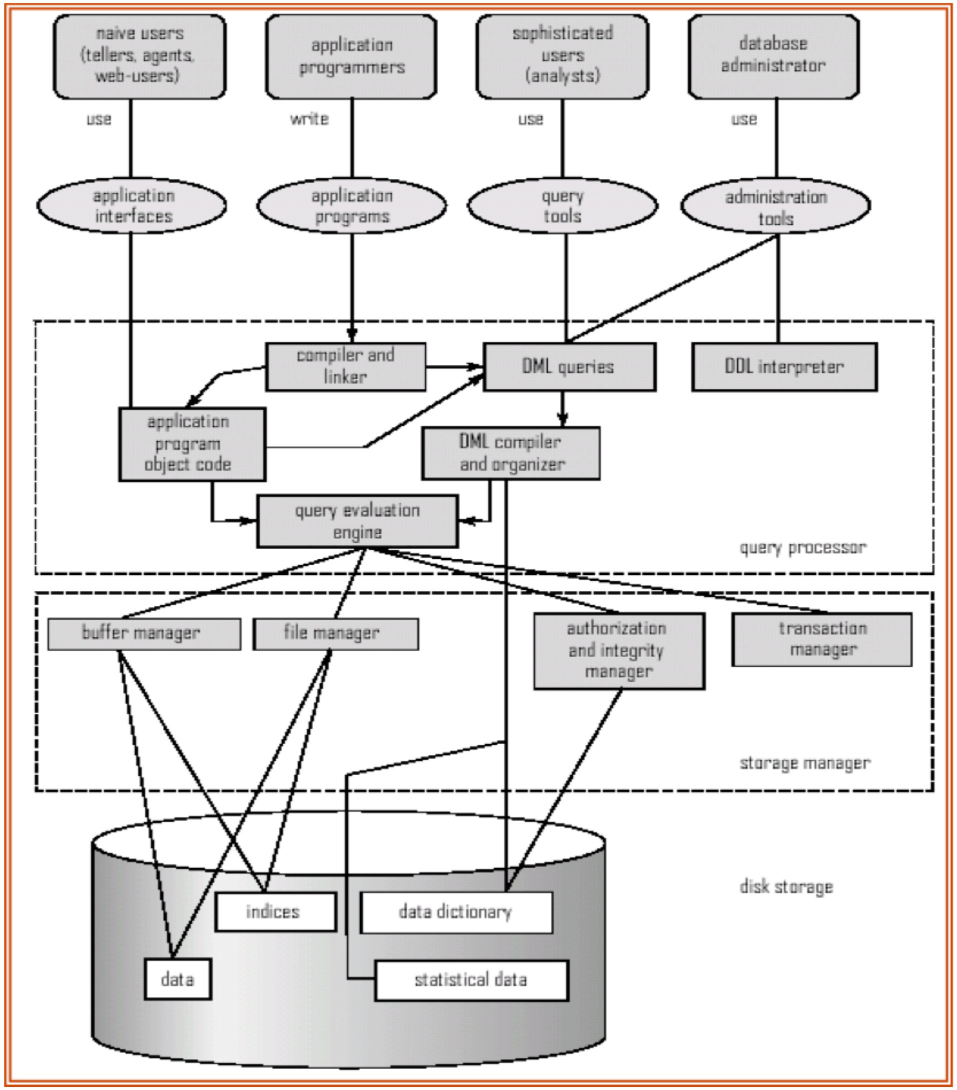
Dr. Wei Wang

Department of Computing

Learning Outcomes

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery

DBMS Components revisited



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 drivers 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 the database.
  - A failure may occur after one of these modifications has 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
  - 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
- **Non-volatile 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
  - usually 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):
    - Write the information onto the first physical block.
    - When the first write successfully completes, write the same information onto the second physical block.
    - The output is completed only after the second write successfully completes.

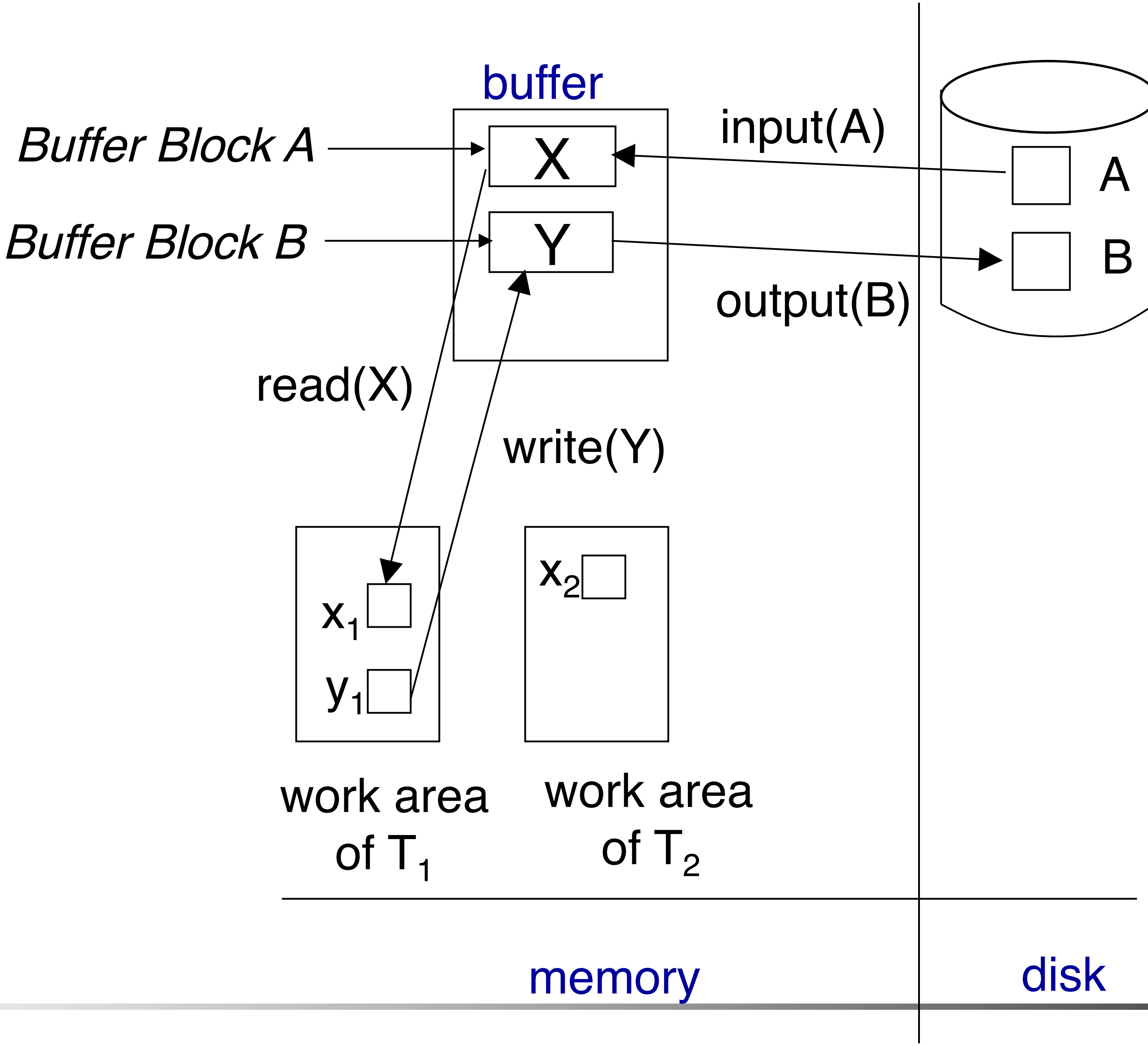
Stable-Storage Implementation cont'd

- 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 (Redundant Arrays of Independent Disks).
  - 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 (or vice verse).

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'd

- 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 of failures, we first output information describing the modifications (e.g., logs) to stable storage without modifying the database itself.
- log-based recovery mechanisms
  - key concepts
  - 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
  $\langle T_i$ **start**$\rangle$ log record
- *Before* T_i executes **write**(X), a log record
  $\langle T_i, X, V_1, V_2\rangle$
  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 $\langle T_i$ **commit**$\rangle$ or $\langle T_i$ **abort**$\rangle$ is written.
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Immediate and Deferred 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* 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 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

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 and Deferred 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 output after T_0 commits

- Note: B_X denotes block containing X.

Concurrency Control and Recovery

- With concurrent transactions, all transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions
- 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. using the* strict two-phase locking protocol.
- Log records of different transactions may be interspersed in the log.

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 (again)
- **Undo and Redo of Transactions**
  - **undo**(T_i) restores the values 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 additional 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 algorithm greatly

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time. Assume that the failure occurs immediately after the last statement.

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

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.

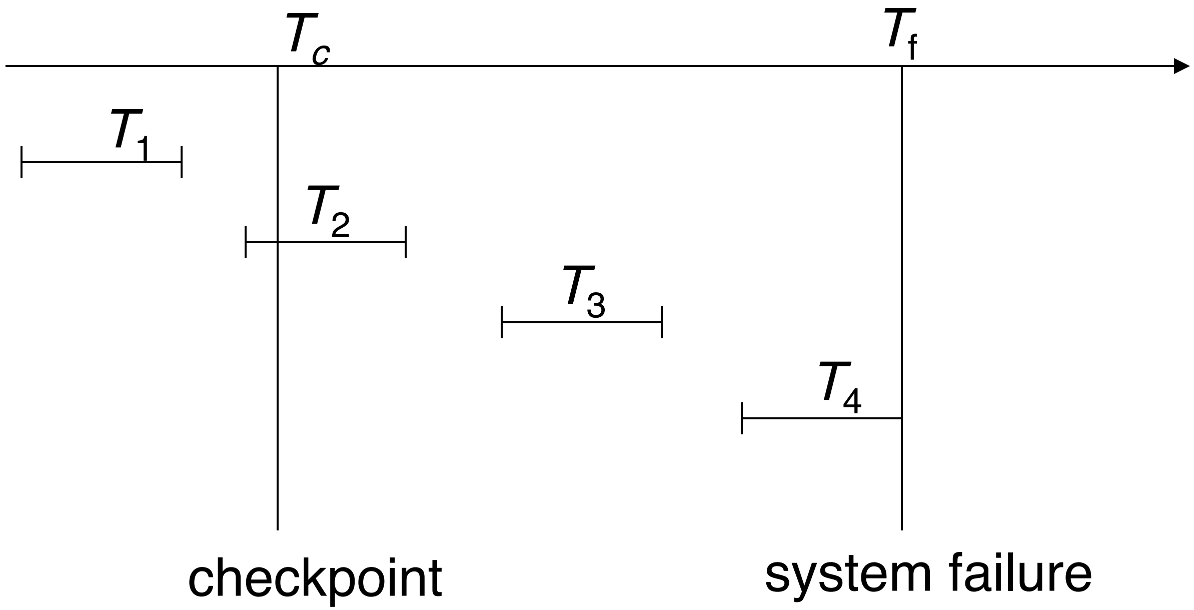
Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  - 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 long time ago.
- Streamline recovery procedure by periodically performing **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 which are active at the time of checkpointing.
  - All updates are stopped while doing checkpointing.

Checkpoints cont'd

- During recovery we need to consider only the most recent transactions that started before the checkpoint but not finished, and transactions that started after checkpoint.
  - Upon failure, 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 (no need to consider them).
- Some earlier part of the logs 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 the 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 (but all instructions in T4 up to the failure point need to be redone)

Recovery Algorithm

- Logging (during normal operation)
  - $\langle T_i$ **start**$\rangle$ at transaction start
  - $\langle T_i, X_j, V_1, V_2\rangle$ for each update, and
  - $\langle T_i$ **commit**$\rangle$ at the end of transaction
- 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 $\langle T_i$ **start**$\rangle$ is found stop the scan and write the log record $\langle T_i$ **abort**$\rangle$

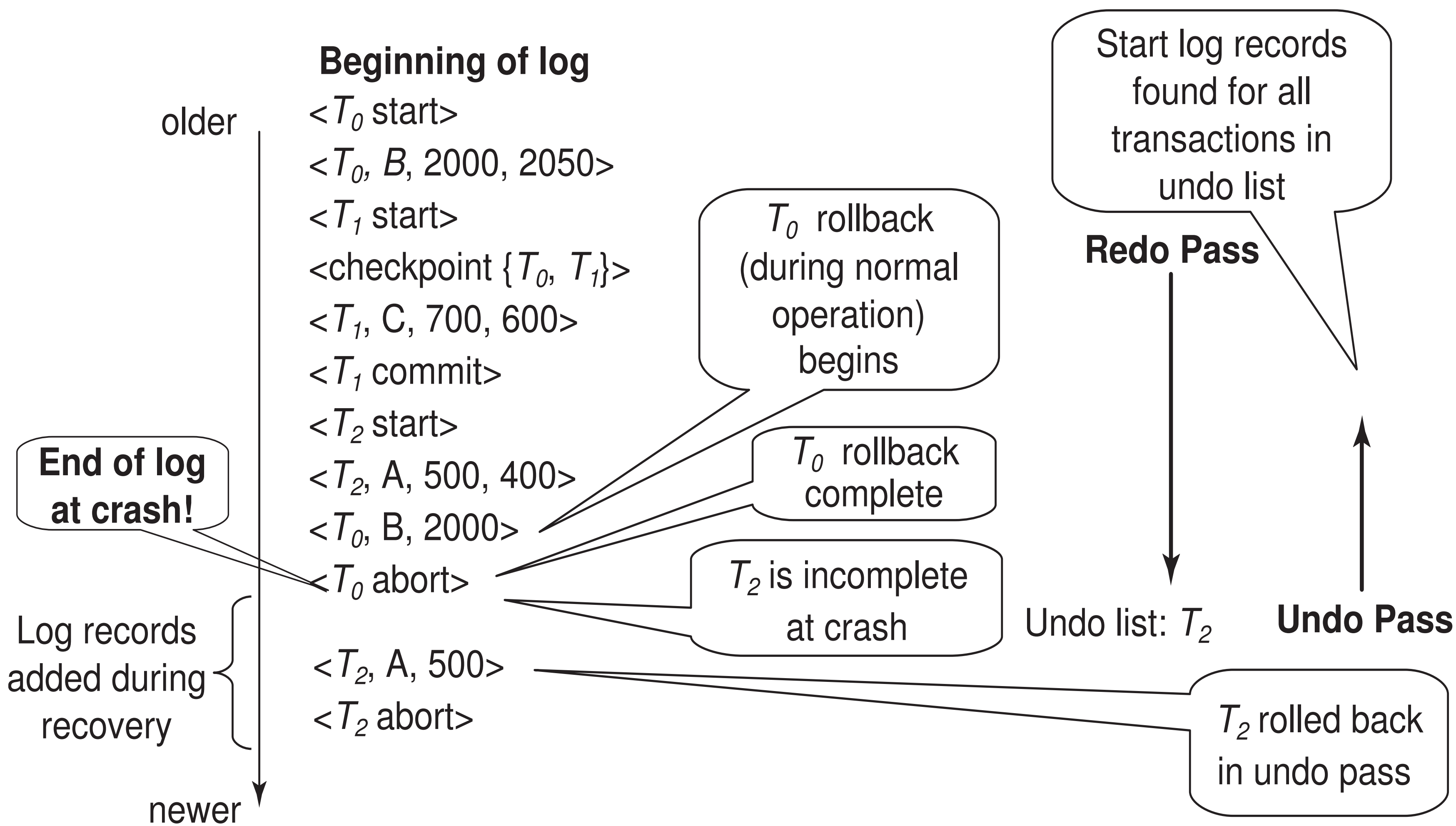
Recovery Algorithm cont'd

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete, at and after checkpoint
  - **Undo phase**: undo all incomplete transactions
- **Redo phase**:
  - Find last <**checkpoint** *L*> record, and set the undo-list to *L* (undo-list = L).
  - Scan forward from above <**checkpoint** *L*> record
    - Whenever a record $<T_i, X_j, V_1, V_2>$ is found, redo it by writing V_2 to X_j
    - Whenever a log record $<T_i$ **start**> is found, add T_i to undo-list
    - Whenever a log record $<T_i$ **commit**> *or* $<T_i$ **abort**> is found, remove T_i from undo-list

Recovery Algorithm cont'd

- **Undo phase:**
  - Scan log backwards from the failure point
    - 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:
      - perform undo by writing V_1 to X_j.
      - write a log record <T_i , X_j, V_1>
    - Whenever a log record <T_i **start**> is found where T_i is in undo-list,
      - Write a log record <T_i **abort**>
      - Remove T_i from undo-list
    - 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 again.

Example of Recovery



End of Lecture

- Summary
  - Failure Classification
  - Storage Structure
  - Recovery and Atomicity
  - Log-Based Recovery

- Reading
  - Textbook 6th edition, chapter 16.1, 16.2, 16.3, 16.4
  - Textbook 7th edition, chapter 19.1, 19.2, 19.3, 19.4