Recovery System

- **Basic Concepts**
- Log-Based Recovery
- Checkpointing
- Recovery Algorithm
- Failure with Loss of Nonvolatile Storage

Recovery System

- Goals:
  - Ensure that ***atomicity*** and ***durability*** properties of transactions are preserved.
  - Restore the database to the consistent state that existed before the failure.

Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error conditions
    - Example: bad input, data not found, overflow, resource limit exceeded
  - **System errors**: the database system must terminate an active transaction due to an error condition
    - Example: deadlock
- **System crash**: a power failure or other hardware or software failure causes the system to crash and the loss of the content of volatile storage.
  - **Fail-stop assumption**: non-volatile storage contents are assumed ***not*** to be corrupted by system crash
    - Well-designed systems have numerous internal checks that bring the system to a halt when there is an error.

Failure Classification (cont.)

- **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
  - Copies of the data on other disks are used to recover from failure.

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

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

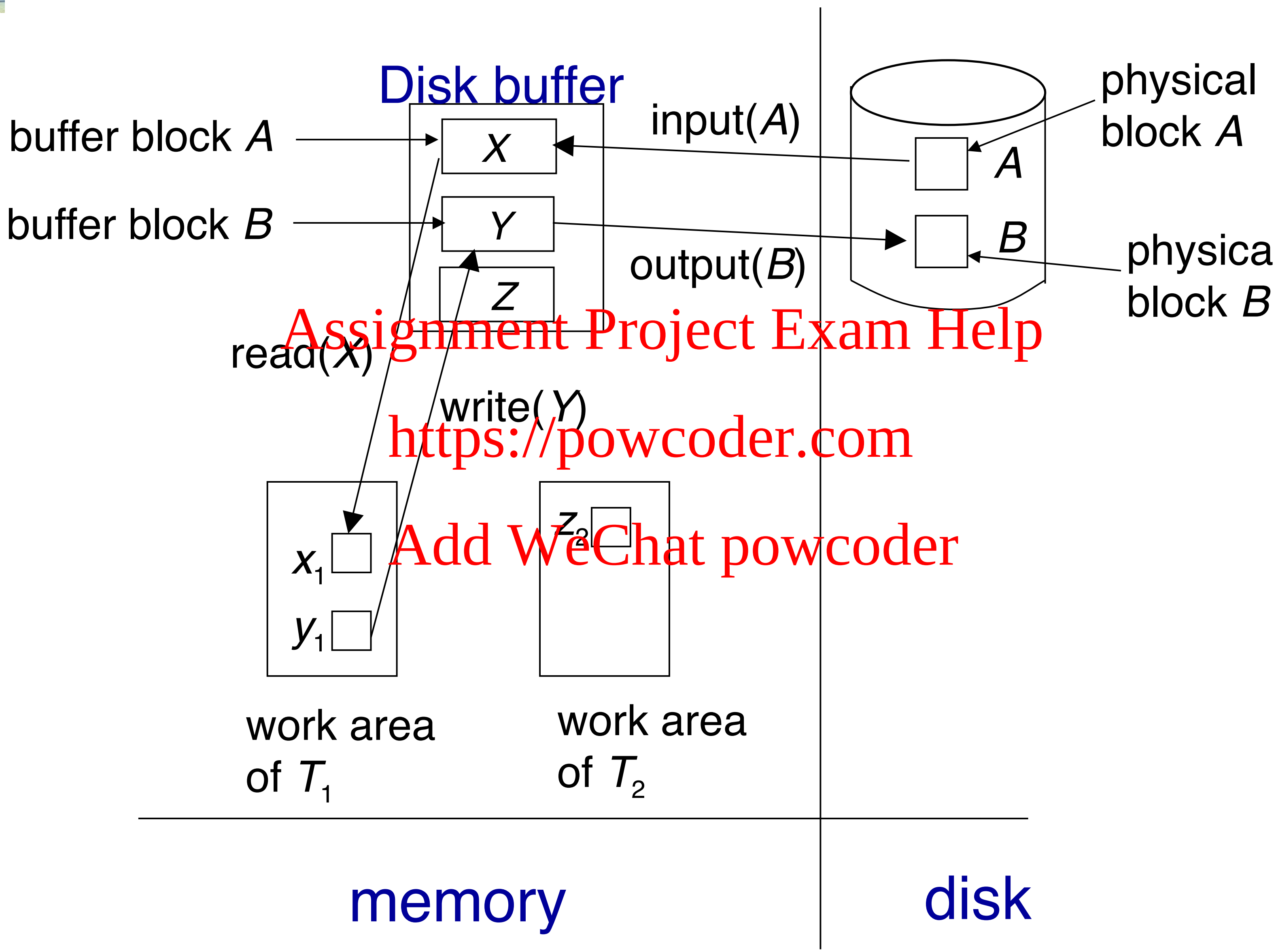
- A database system
  - resides permanently on nonvolatile storage such as disks, and
  - parts of the database are in memory at any time.

- **Physical blocks** are those blocks residing on the disk.

- **Buffer blocks** are the blocks residing temporarily in main memory in an area called ***disk buffer***.

- 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.

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 is done by:
  - **read**(X) assigns the value of data item X to the local variable x_i; need to issue **input**(B_X) if B_X in not in main memory (B_X: block containing X).
  - **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

- *Reminder*: **Atomicity**: Either all operations of the transaction are reflected properly in the database, or none are.
- To ensure atomicity despite failures, we first output information describing the modifications (**log**) to stable storage ***before*** modifying the database itself.
  - Ensure that all modifications performed by committed transactions are reflected in the database.
  - Ensure that no modifications made by an aborted transaction persist in the database.
- We study **log-based recovery mechanisms** in detail
  - We first present key concepts
  - And then present the actual recovery algorithm

Recovery System

- Basic Concepts
- **Log-Based Recovery**
- Checkpointing
- Recovery Algorithm
- Failure with Loss of Nonvolatile Storage

Log-Based Recovery

- A **log** is a sequence of **log records**, which keep information about update activities on the database.
  - The **log** is kept on stable storage
- When transaction T_i starts, it registers itself by writing a
  <T_i start> log record
- ***Before*** T_i executes **write**(X), an ***update 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 its last statement, the log record **<T_i commit >** is written.
- When T_i has aborted, the log record **<T_i abort >** is written.
- We assume that every log record is output directly to stable storage once created.

Immediate Database Modification

- To understand the role of log records in recovery, we need to consider the steps a transaction takes in modifying a data item.
  - The transaction performs some computations in its own private work area.
  - The transaction modifies the corresponding data block in the disk buffer.
  - The database system executes the output operation that writes the data block to disk.
- A transaction ***modifies the database*** if it performs an update on a disk buffer ***or*** on the disk itself.
- If a transaction is allowed to modify the database before it commits, we call this **immediate-modification**.
- ***Update log record*** must be written ***before*** modifying the database.
- Output of updated blocks to disk 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.

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. So, a recovery algorithm must consider the following possibilities.
  - A transaction may have modified the database ***before*** it commits and, as a result of a subsequent failure, may need to abort.
  - A transaction may have committed although some of its database modifications exist ***only in the disk buffer*** and ***not on disk***.

Immediate 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

Output B_C before T_1 commits

Output B_A and B_B after T_0 commits

- Note: B_X denotes block containing X.

Concurrency Control and Recovery

- 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 to 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)

- 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.
  - 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
- **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_1 a special ***redo-only*** log record $<T_i, X, V_1>$ 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 (the redo-only log records)
    - 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.

<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>
(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 <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.

Recovery System

- Basic Concepts
- Log-Based Recovery
- **Checkpointing**
- Recovery Algorithm
- Failure with Loss of Nonvolatile Storage

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**

  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

Checkpoints (Cont.)

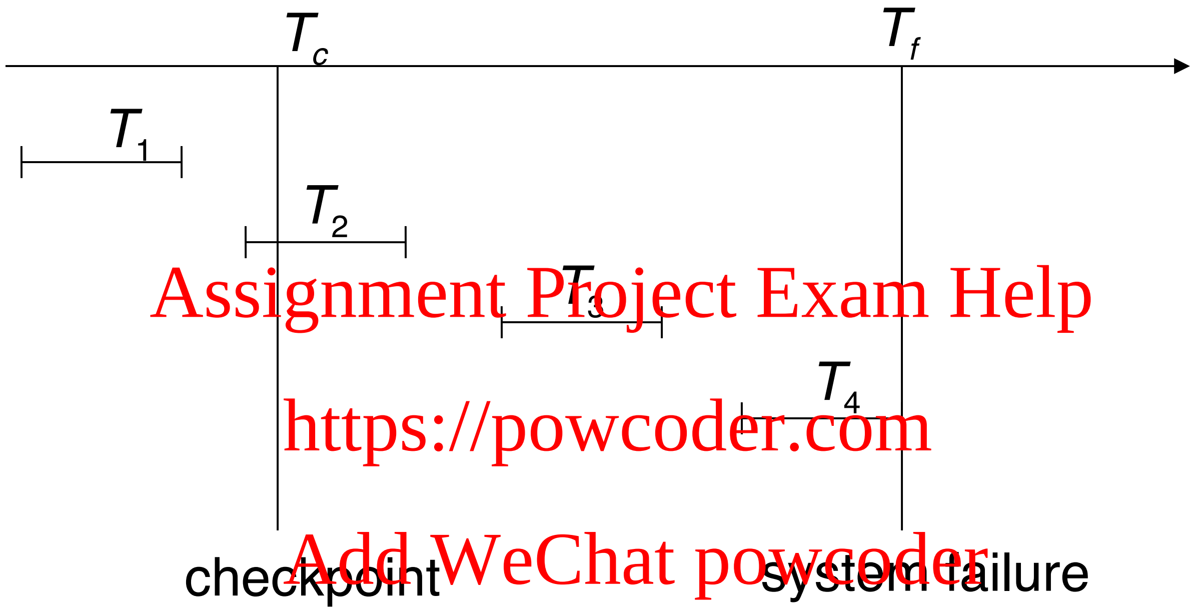
- Transactions that committed or aborted before the checkpoint must have written their updates to the database either
  - prior to the checkpoint or
  - as part of the checkpoint itself.
- So, there is no need to consider these transactions at recovery time.

- During recovery
  1. Scan backwards from end of log to find the most recent <**checkpoint** *L*> record

  2. Only transactions that are in *L* or started after the checkpoint need to be redone or undone

- Some earlier part of the log may be needed for undo operations
  1. Continue scanning backwards till a record <T_i **start**> is found for every transaction T_i in *L*.
  2. 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

Recovery System

- Basic Concepts
- Log-Based Recovery
- Checkpointing
- **Recovery Algorithm**
- Failure with Loss of Nonvolatile Storage

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 redo-only log records are also 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 (**repeating history**)
  - **Undo phase**: undo all incomplete transactions

- **Redo phase**:
  - Find last <**checkpoint** *L*> record, and set undo-list to *L*.

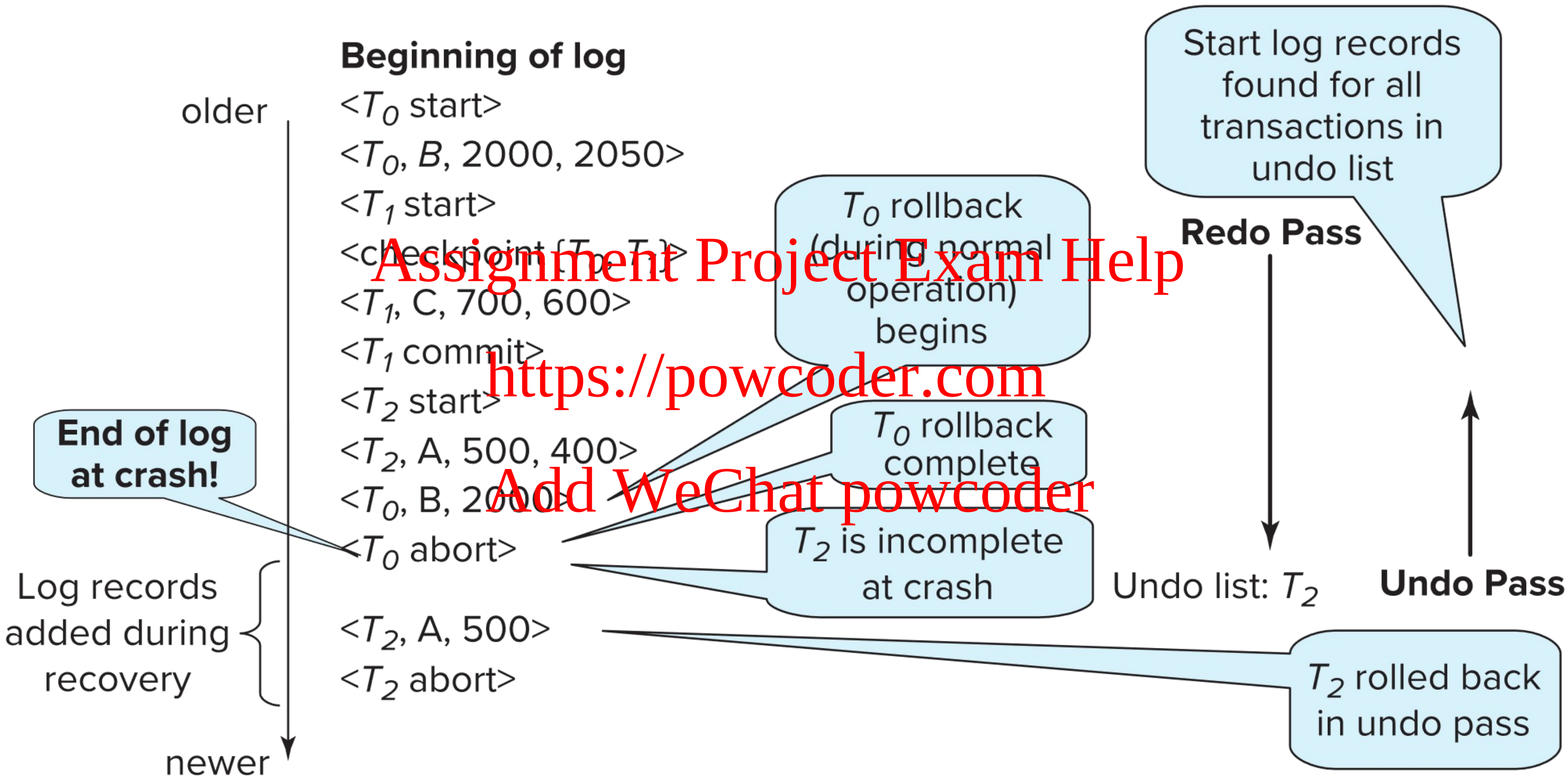
  - Scan forward from the <**checkpoint** *L*> record

    1. Whenever a record $<T_i, X_j, V_1, V_2>$ or $<T_i, X_j, V_2>$ 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:**
  - 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 undo by writing V_1 to X_j.
       - 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,
       - Write a log record $<T_i$ **abort**$>$
       - 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 resume.

Example of Recovery



Recovery System

- Basic Concepts
- Log-Based Recovery
- Checkpointing
- Recovery Algorithm
- **Failure with Loss of Nonvolatile Storage**

Failure with Loss of Nonvolatile Storage

- *Reminder*: 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 residing 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.

Recovering 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 to bring the database to the most recent consistent state.

More….

- In addition to ***immediate modifications***, there are ***deferred modifications***: all the write operations of a transaction are deferred until the transaction has been committed.

- In addition to the popular ***no-force*** policy: a transaction is allowed to commit even if its updated blocks have not yet been written back to disk, there is ***force*** policy: transactions would force-output all modified buffer blocks to disk when they commit.

- In addition to the popular ***steal*** policy: the system is allowed to write blocks containing updates of uncommitted transactions to disk, there is ***no-steal*** policy: blocks modified by a transaction that is still active should not be written to disk.

More….

- Although it has been assumed that every log record is output directly to stable storage once created, log records may be buffered in main memory temporarily before output to stable storage, under additional requirements (because such log records are lost if the system crashes).

- Although it has been assumed that all updates to the database be temporarily suspended while the checkpoint is in progress, updates may be permitted during checkpointing but we need to deal with incomplete checkpoints when the system crashes before checkpoint is done.