

DATABASE MANAGEMENT SYSTEMS

Module - V

Recovery System



Outline

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



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

- Suppose transaction T_i 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



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 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
 - See book for more details on how to implement stable storage



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):
 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.



Protecting storage media from failure (Cont.)

- Copies of a block may differ due to failure during output operation.
- To recover from failure:
 1. First find inconsistent blocks:
 1. *Expensive solution*: Compare the two copies of every disk block.
 2. *Better solution*:
 - Record in-progress disk writes on non-volatile storage (Flash, 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 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.

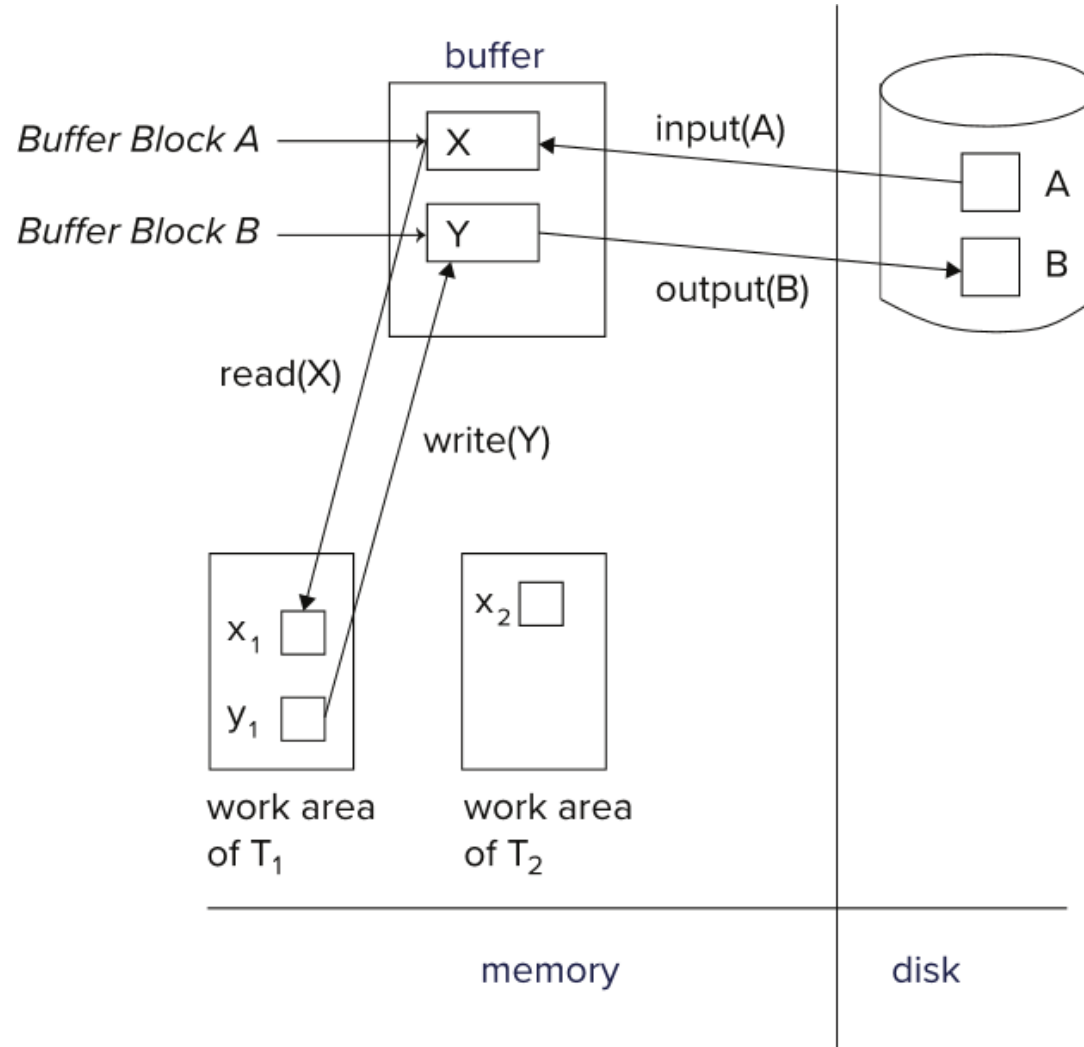


Data Access (Cont.)

- 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



Example of Data Access

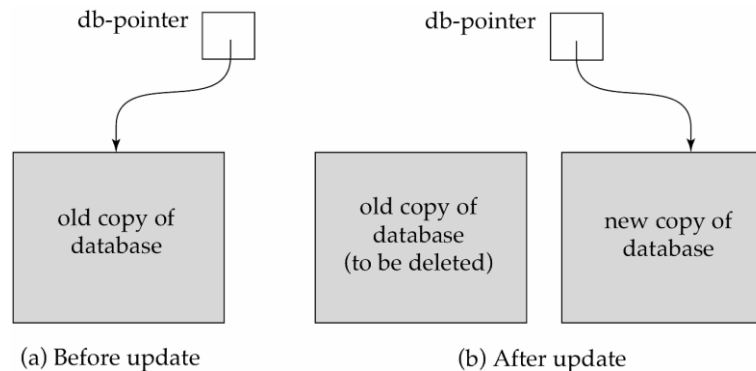




Recovery and Atomicity

- 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-copy** and **shadow-paging** (brief details in book)

shadow-copy





Log-Based Recovery

- A **log** is a sequence of **log records**. The records 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 $\langle T_i \text{ start} \rangle$ 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 its last statement, the log record $\langle T_i \text{ commit} \rangle$ is written.
- Two approaches using logs
 - Immediate database modification
 - Deferred database modification.



Immediate 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
 - (Will see later that how to postpone log record output to some extent)
- 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.
- 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 Database Modification Example

Log	Write	Output
-----	-------	--------

<T₀ start>

<T₀, A, 1000, 950>

<T₀, B, 2000, 2050>

A = 950
B = 2050

<T₀ commit>

<T₁ start>

<T₁, C, 700, 600>

C = 600

B_B, B_C

B_C output before T₁ commits

<T₁ commit>

B_A

B_A output after T₀ 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., 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)
- Log records of different transactions may be interspersed in the log.



Undo and Redo Operations

■ 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 $\langle T_i, X, V \rangle$ is written out
 - When undo of a transaction is complete, a log record $\langle T_i, \mathbf{abort} \rangle$ is written out.
- **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



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



Recovering from Failure (Cont.)

- Suppose that transaction T_i was undone earlier and the $\langle T_i$ **abort** \rangle record was written to the log, and then a failure occurs,
- On recovery from failure transaction T_i is redone
 - Such a **redo** redoes all the original actions of transaction T_i *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.

$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
$\langle T_0, A, 1000, 950 \rangle$	$\langle T_0, A, 1000, 950 \rangle$	$\langle T_0, A, 1000, 950 \rangle$
$\langle T_0, B, 2000, 2050 \rangle$	$\langle T_0, B, 2000, 2050 \rangle$	$\langle T_0, B, 2000, 2050 \rangle$
	$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$
	$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$
	$\langle T_1, C, 700, 600 \rangle$	$\langle T_1, C, 700, 600 \rangle$
		$\langle T_1 \text{ commit} \rangle$
(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**
 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.
 4. All updates are stopped while doing checkpointing

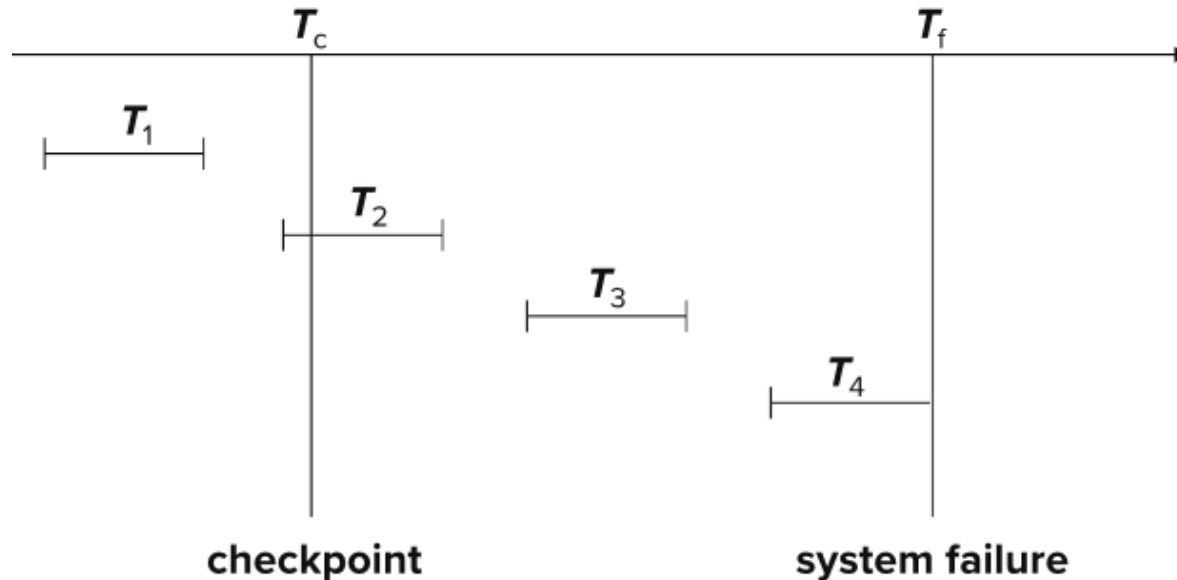


Checkpoints (Cont.)

- 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.



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
- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone



Recovery Algorithm



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



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



Recovery Algorithm (Cont.)

- **Undo phase:**
 1. Scan log backwards from end
 1. Whenever 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:
 1. perform undo by writing V_1 to X_j .
 2. write a log record $\langle T_i, X_j, V_1 \rangle$
 2. Whenever a log record $\langle T_i \text{ start} \rangle$ is found where T_i is in undo-list,
 1. Write a log record $\langle T_i \text{ abort} \rangle$
 2. Remove T_i from undo-list
 3. Stop when undo-list is empty
 1. 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



Example of Recovery

Beginning of log

older
↓
< T_0 start>
< $T_0, B, 2000, 2050$ >
< T_1 start>
<checkpoint { T_0, T_1 }>
< $T_1, C, 700, 600$ >
< T_1 commit>
< T_2 start>
< $T_2, A, 500, 400$ >
< $T_0, B, 2000$ >
< T_0 abort>
< $T_2, A, 500$ >
< T_2 abort>
↓
newer

End of log at crash!

Log records added during recovery

T_0 rollback (during normal operation) begins

T_0 rollback complete

T_2 is incomplete at crash

Start log records found for all transactions in undo list

Redo Pass

Undo list: T_2

Undo Pass

T_2 rolled back in undo pass



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 (Cont.)

- The rules below must be followed if log records are buffered:
 - Log records are output to stable storage in the order in which they are created.
 - 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
 - Strictly speaking, WAL only requires undo information to be output



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 commit
- 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 with undo information 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



Buffer Management (Cont.)

- Database buffer can be implemented either
 - In an area of real main-memory reserved for the database, or
 - In virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
 - Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
 - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.



Buffer Management (Cont.)

- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
 - When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
 - When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
 - Known as **dual paging** problem.
 - Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
 1. Output the page to database instead of to swap space (making sure to output log records first), if it is modified
 2. Release the page from the buffer, for the OS to use
 - Dual paging can thus be avoided, but common operating systems do not support such functionality.



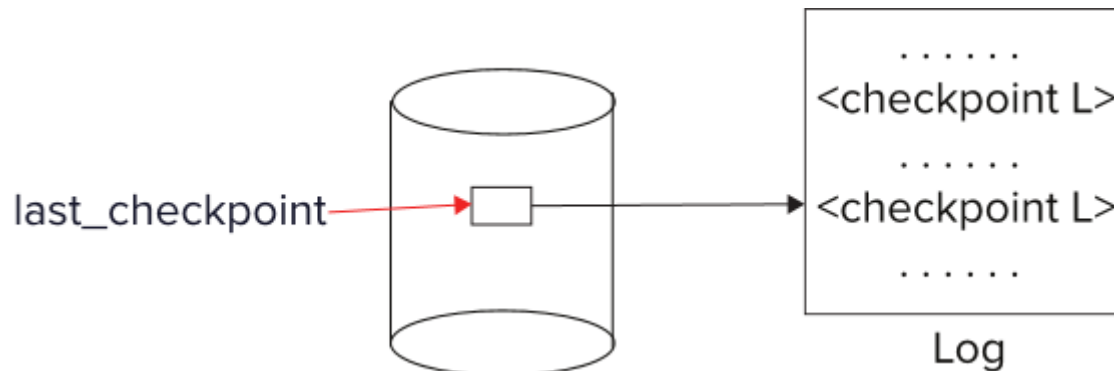
Fuzzy Checkpointing

- 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





Failure with Loss of Nonvolatile 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 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
- Can be extended to allow transactions to be active during dump; known as **fuzzy dump** or **online dump**
 - Similar to fuzzy checkpointing