Unit 6: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Recovery Algorithms
- Buffer Management
- Failure with loss of non-volatile storage

Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition such as bad input, data not found, overflow, etc.
  - **System errors**: The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution.
- **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 failure during data transfer destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

Storage Structure

- **Volatile storage**:
  - does not survive power failure, system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**:
  - survives power failure, 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

- To implement stable storage, we need to replicate the needed information in several non volatile storage media (usually disk) and to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.
- More secure systems keep a copy of each block of stable storage at a remote site, writing it out over a computer network, in addition to storing the block on a local disk system.
- Since the blocks are output to a remote system as and when they are output to local storage, once an output operation is complete, the data is not lost, even in the event of a disaster such as a fire or flood.
- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.

Stable-Storage Implementation

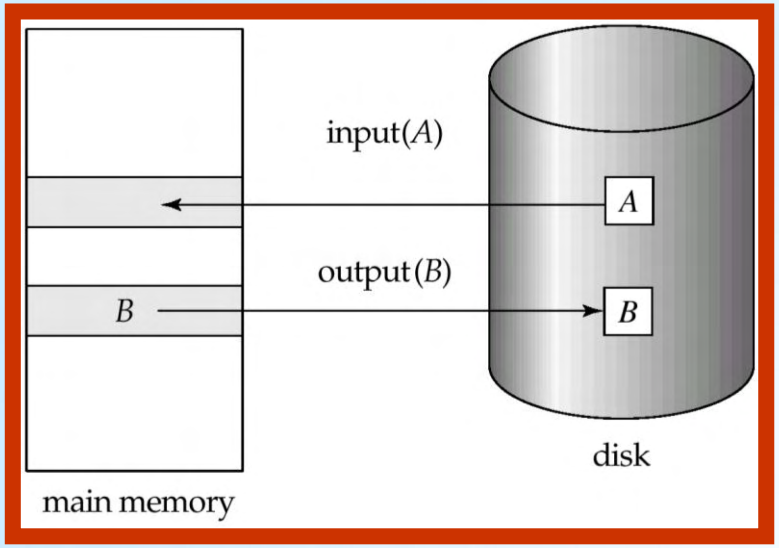
- Failure during data transfer at remote site can still result in inconsistent copies: Block transfer can result in
  - **Successful completion**. The transferred information arrived safely at its destination.
  - **Partial failure**. A failure occurred in the midst of transfer, and the destination block has incorrect information.
  - **Total failure**. The failure occurred sufficiently early during the transfer that the destination block remains intact.
- Protecting storage media from failure during data transfer (one solution): The system maintain two physical blocks for each logical database block.
- 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.

Stable-Storage Implementation (Cont.)

- If the system fails while blocks are being written, it is possible that the two copies of a block are inconsistent with each other.
- During recovery, for each block, the system would need to examine two copies of the blocks. If both are the same and no detectable error exists, then no further actions are necessary.
- If the system detects an error in one block, then it replaces its content with the content of the other block.
- If both blocks contain no detectable error, but they differ in content, then the system replaces the content of the first block with the value of the second.
- This recovery procedure ensures that a write to stable storage either **succeeds completely** (that is, updates all copies) or results in **no change.**

Data Access

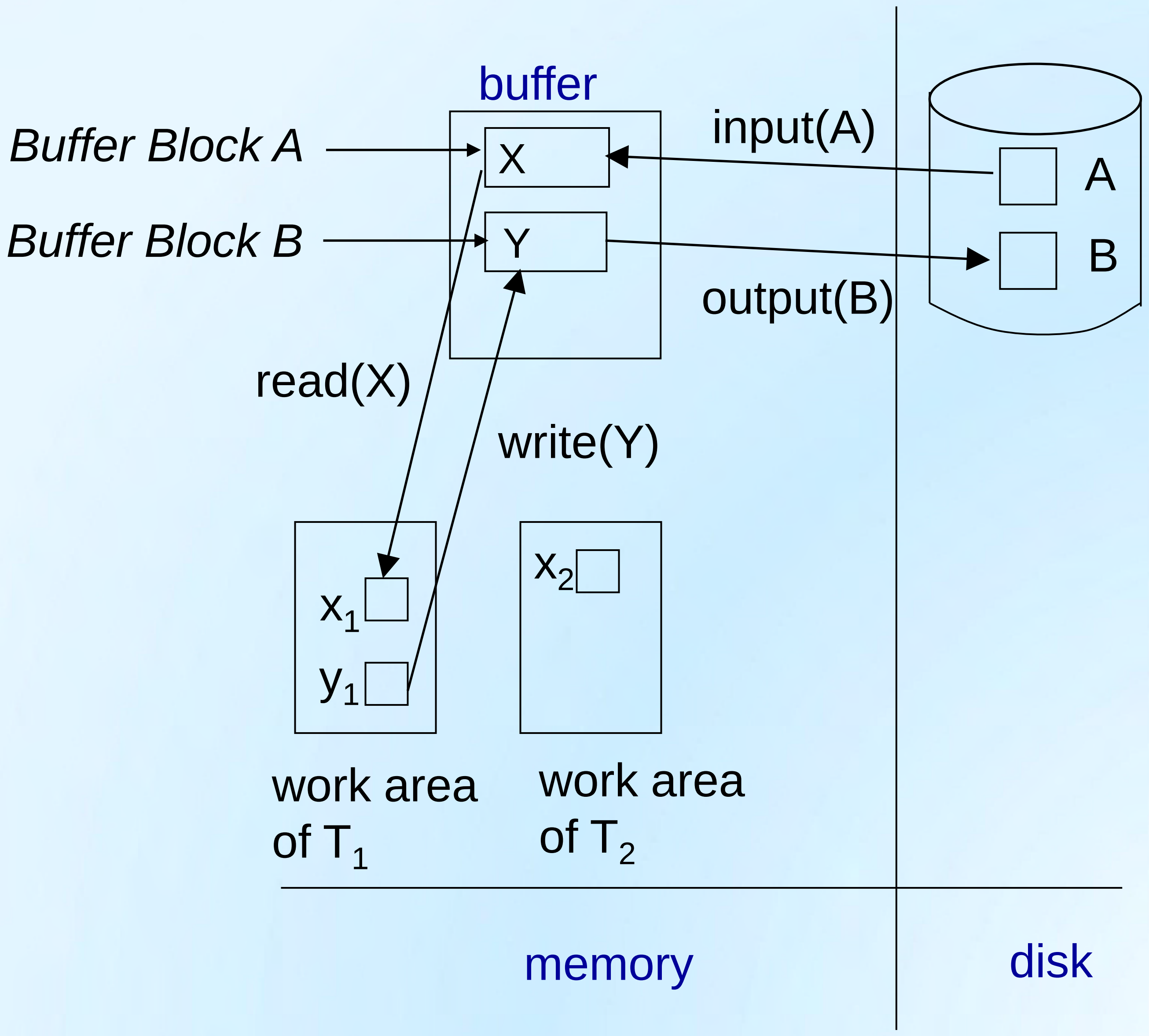
- the database system resides permanently on nonvolatile storage (usually disks) and only parts of the database are in memory at any time.
- Database is partitioned into fixed-length storage units called **blocks.**
- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- **Disk buffer** area of main memory where blocks resides temporarily.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input**(*A*) transfers the physical block *A* from disk to main memory.
  - **output**(*B*) transfers the buffer block *B* from main memory 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.
- The system creates this work area when the transaction is initiated; the system removes it when the transaction either commits or aborts.
  - 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.
- 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

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Consider transaction T_i that transfers $50 from account *A* to account *B*;.
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- Goal is either to perform all database modifications made by T_i or none at all
- Several output operations may be required for T_i (to output *A* and *B*). A failure may occur after one of these modifications have been made but before all of them are made.

Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications (**logs**) to stable storage without modifying the database itself.
- **log-based recovery mechanisms:**
  - Deferred Database Modification
  - Immediate Database Modification
- Less used alternative: **shadow-paging**

Log-Based Recovery

- **Log** is the most widely used structure for recording database modifications.
- The log is a sequence of **log records**, and maintains a record of update activities on the database.
- A **log** is kept on stable storage.
- It has these fields:
  - Transaction Identifier
  - Data-item Identifier
  - Old Value
  - New Value

Log-Based Recovery

- 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.
- We assume for now that log records are written directly to stable storage.

Deferred Database Modification

- The **deferred database modification** scheme ensures atomicity by recording all modifications to the log, but defers all the **write** operations of a transaction until transaction partial commit.
- Assume that transactions execute serially.
- Transaction starts by writing <T_i ***start***> record to log.
- A **write**(*X*) operation results in a log record <T_i, *X*, *V*> being written, where *V* is the new value for *X*
  - Note: old value is not needed for this scheme
- The write is not performed on *X* at this time, but is deferred.
- When T_i partially commits, <T_i **commit**> is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.

Deferred Database Modification (Cont.)

- During recovery after a crash, a transaction needs to be redone if and only if both <T_i **start**> and<T_i **commit**> are there in the log.
- Redoing a transaction T_i (**redo**T_i) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while
  - the transaction is executing the original updates, or
  - while recovery action is being taken
- example transactions T_0 and T_1 (T_0 executes before T_1):

T_0: **read** (*A*)

A: - A - 50

Write (*A*)

read (*B*)

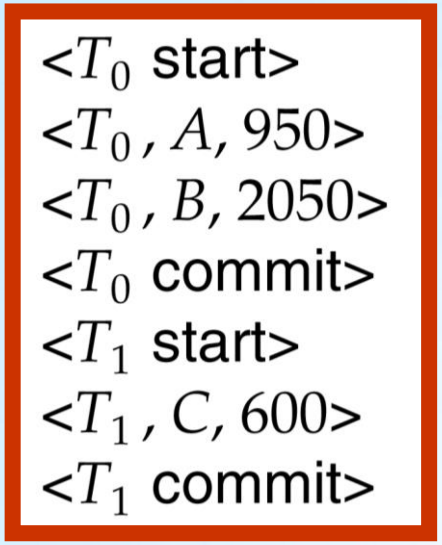
B:- B + 50

write (*B*)

T_1 : **read** (*C*)

C:- C- 100

write (*C*)



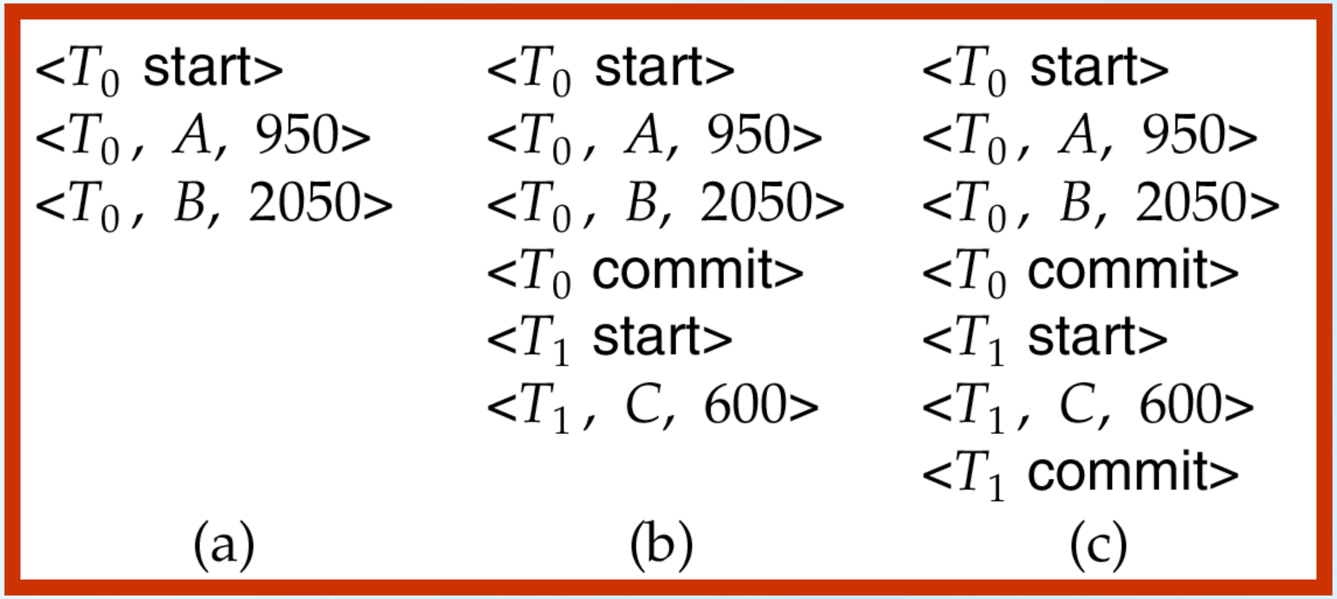
<T_0 start>
<T_0, A, 950>
<T_0, B, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 600>
<T_1 commit>

State of the Log and Database Corresponding to T_0 and T_1

Log	Database
<T_0 start>	
<T_0, A, 950>	
<T_0, B, 2050>	
<T_0 commit>	
	$A = 950$
	$B = 2050$
<T_1 start>	
<T_1, C, 600>	
<T_1 commit>	
	$C = 600$

Deferred Database Modification (Cont.)

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

- If log on stable storage at time of crash is as in case:

  (a) No redo actions need to be taken

  (b) redo(T_0) must be performed since <T_0 **commi**t> is present

  (c) **redo**(T_0) must be performed followed by redo(T_1) since
  <T_0 **commit**> and <T_i commit> are present

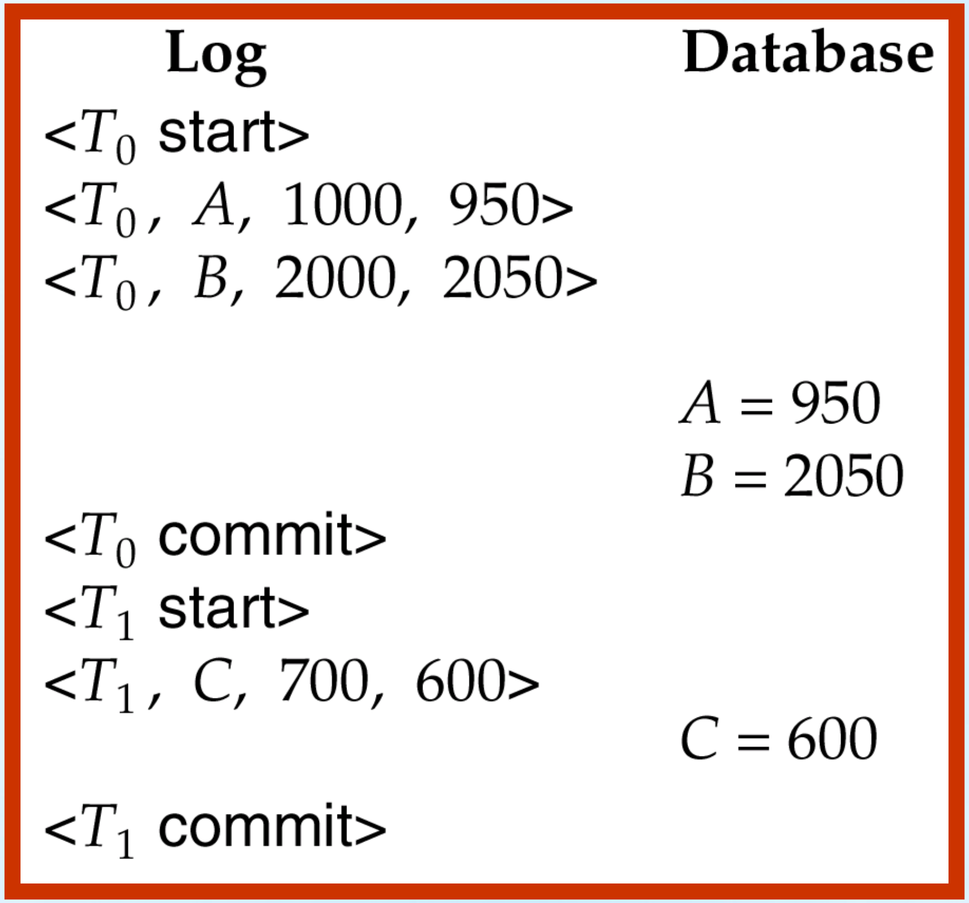
Immediate Database Modification

- The **immediate database modification** scheme allows database modifications to be output to the database while transaction is still in active state (uncommitted modification).
- since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
  - ★ We assume that the log record is output directly to stable storage
  - ★ Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage
- Output of updated blocks 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.

Portion of the System Log Corresponding to T_0 and T_1

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

State of System Log and Database Corresponding to T_0 and T_1



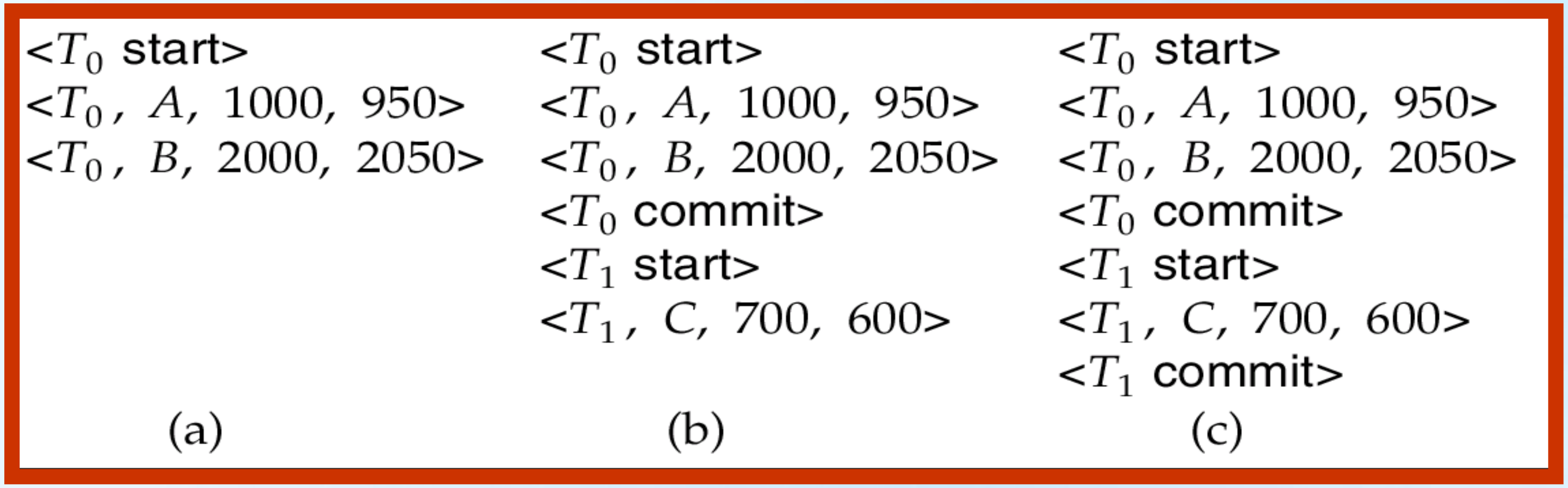
Log	Database
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	$A = 950$
	$B = 2050$
<T_0 commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	$C = 600$
<T_1 commit>	

Immediate Database Modification (Cont.)

- Recovery procedure has two operations instead of one:
  - **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
  - **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
- Both operations must be **idempotent**
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery
- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
  - Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.
- Undo operations are performed first, then redo operations.

Immediate DB Modification Recovery Example

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



$<T_0$ start$>$
$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$

(a)

$<T_0$ start$>$
$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$
$<T_0$ commit$>$
$<T_1$ start$>$
$<T_1, C, 700, 600>$

(b)

$<T_0$ start$>$
$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$
$<T_0$ commit$>$
$<T_1$ start$>$
$<T_1, C, 700, 600>$
$<T_1$ commit$>$

(c)

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

(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

- When a system crash occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone.
- In principle, we need to search the entire log to determine this information.
- 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 check pointing.

Checkpoints (Cont.)

- After a failure has occurred, the system examines the log to find the last <*checkpoint L*> record.
- This can be done by searching the log backward, from the end of the log, until the first <*checkpoint L*> record is found.
- The redo or undo operations need to be applied only to transactions in *L, and* to all transactions that started execution after the <*checkpoint L*> *record was* written to the log.
- Let us denote this set of transactions as *T.*

Checkpoints (Cont.)

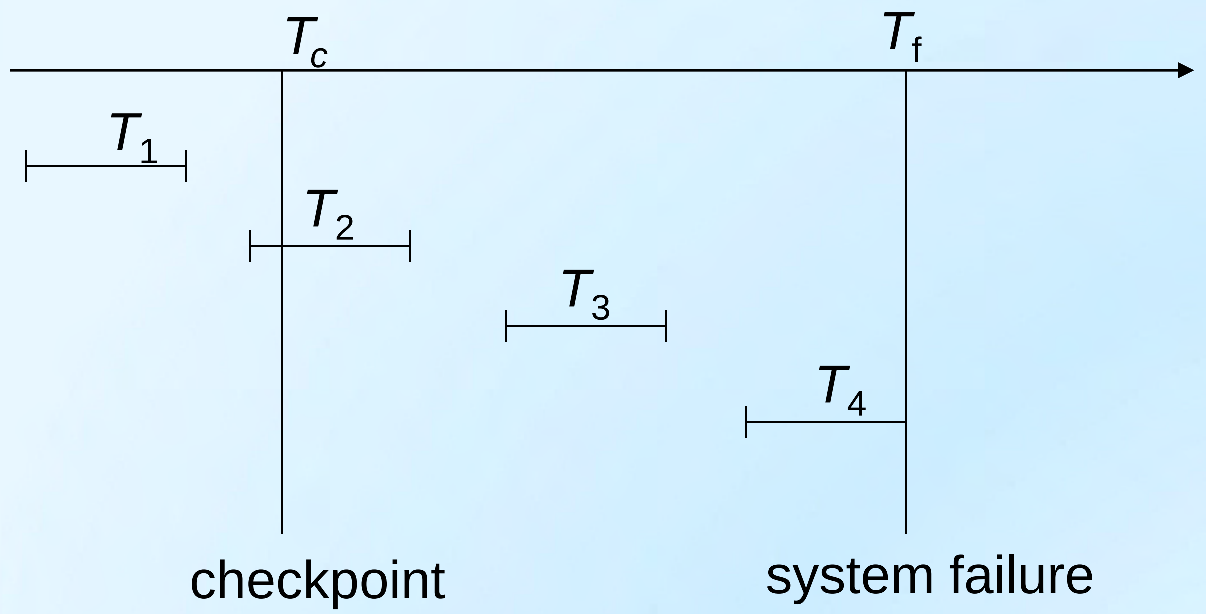
- Immediate Database Modification:

  1. For all transaction T_k in T that have no < T_k commit > record in the log, execute undo (T_k) .

  2. For all transaction T_k in T that have < T_k commit > record in the log, execute redo (T_k) .

- Deferred Database Modification:
  - Apply redo operation only.

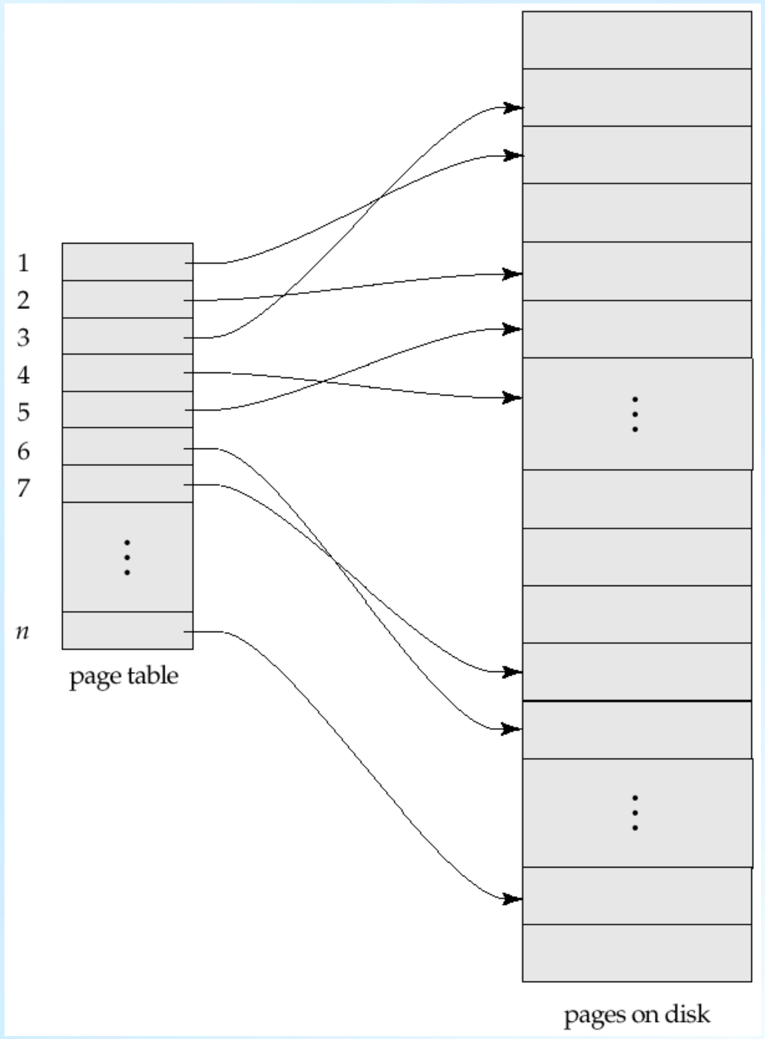
Example of Checkpoints



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

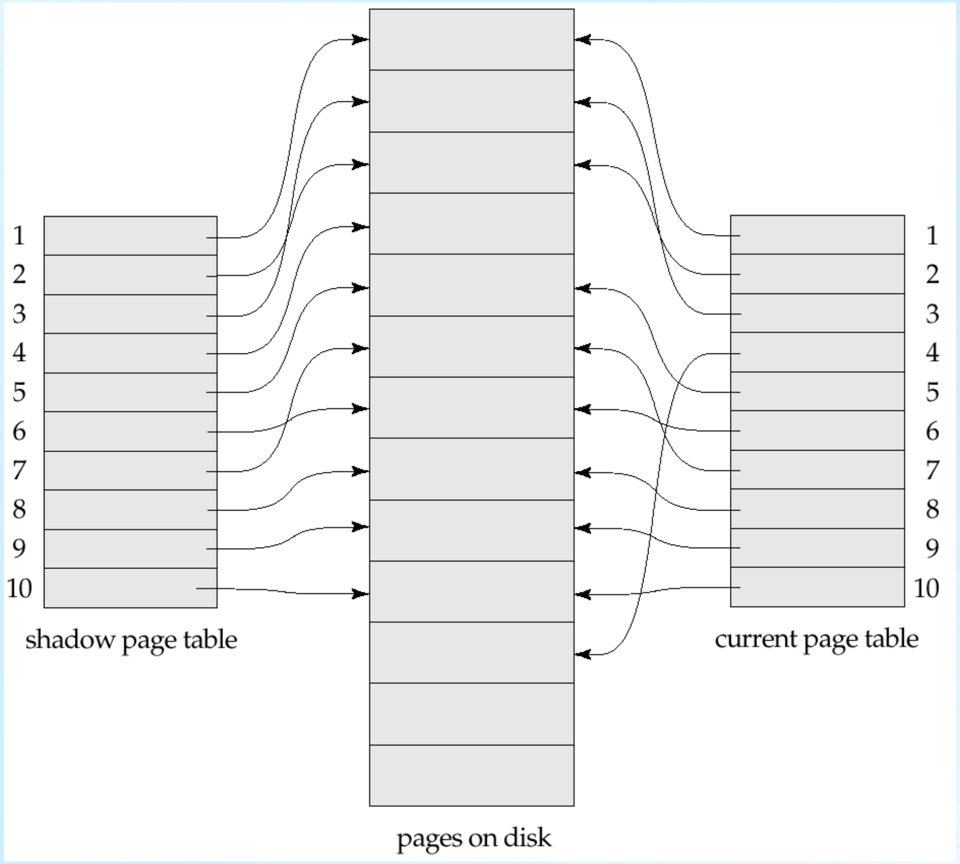
Shadow Paging

- **Shadow paging** is an alternative to log-based recovery; this scheme is useful if transactions execute serially
- Idea: maintain *two* page tables during the lifetime of a transaction – 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
- To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.



Example of Shadow Paging

- Whenever any page is about to be written for the first time
  - First finds an unused page on disk.
  - A copy of the page to be modified 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



Shadow and current page tables after write to page 4

Shadow Paging (Cont.)

- To commit a transaction :
  1. Flush all modified pages in main memory to disk
  2. Output current page table to disk
  3. Make the current page table the new shadow page table, as follows:
     - keep a pointer to the shadow page table at a fixed (known) location on disk.
     - to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
- Once pointer to shadow page table has been written, transaction is committed.
- No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
- Pages not pointed to from current/shadow page table should be freed (garbage collected).

Shadow Paging (Cont.)

- Advantages of shadow-paging over log-based schemes
  - no overhead of writing log records
  - recovery from crashes is faster (no undo or redo).
- Disadvantages :
  - Copying the entire page table is very expensive
    - Can be reduced by using a page table structured like a B^+-tree
      - No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
  - Commit overhead is high even with above extension
    - Need to flush every updated page, and page table
  - Data gets fragmented (related pages get separated on disk)
  - After every transaction completion, the database pages containing old versions of modified data need to be garbage collected
  - Hard to extend algorithm to allow transactions to run concurrently
    - Easier to extend log based schemes

Recovery Algorithm

- We are now ready to present the full recovery algorithm using log records for recovery from transaction failure and a combination of the most recent checkpoint and log records to recover from a system crash.
- A data item that has been updated by an uncommitted transaction cannot be modified by any other transaction, until the first transaction has either committed or aborted.

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**>

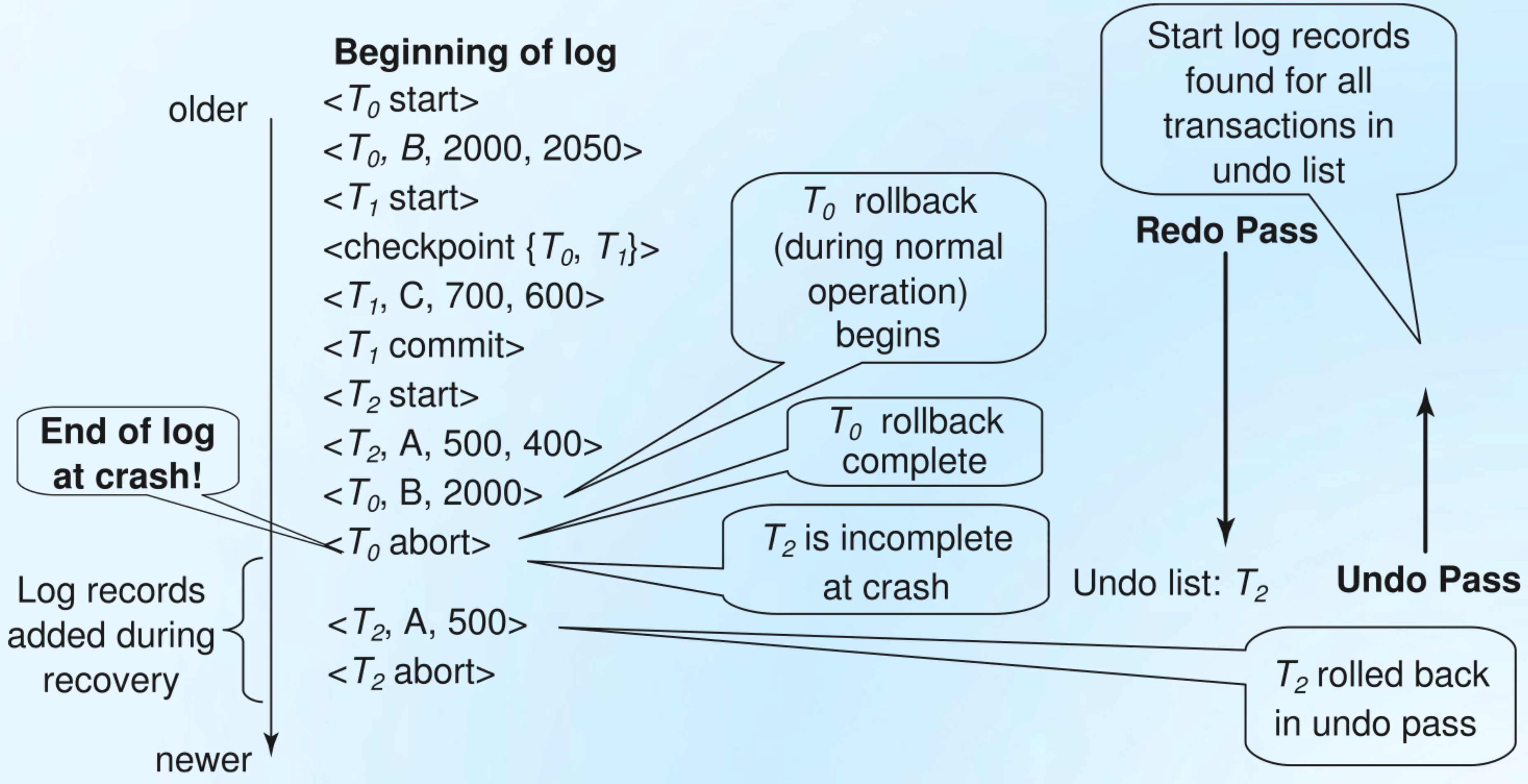
Recovery Algorithm (Cont.)

- **Recovery from failure (After a system Crash)**: 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> *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
- At the end of the redo phase, undo-list contains the list of all transactions that are incomplete, that is, they neither committed nor completed rollback before the crash.

Recovery Algorithm (Cont.)

- **Undo phase:** In the **undo phase,** the system rolls back all transactions in the undo-list.
  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> **compensation log records**
     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



Log Record Buffering

- Output every log record to stable storage at the time it is created, imposes a high overhead on system execution.
- **So** it is desirable to output multiple log records at once.
- **Log record buffering**: write log records to a log buffer in main memory, where they stay temporarily until they are output to stable storage.
- Multiple log records can be gathered in the log buffer and output to stable storage in a single output operation, reducing the I/O cost.
- The order of log records in the stable storage must be exactly the same as the order in which they were written to the log buffer.

Log Record Buffering (Cont.)

- A log record may reside in main memory (volatile storage) for a considerable time before it is output to stable storage.
- Since such log records are lost if the system crashes, we must impose additional requirements on the recovery techniques to ensure transaction atomicity:
  - Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage.
  - Before the <T_i **commit**> log record can be output to stable storage, all log records pertaining to the transaction T_i must have 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

Database Buffering

- Some database blocks are kept in memory buffer (buffer 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
- **force policy:** Transactions would force-output all modified blocks to disk when they commit**.**
  - More expensive commit
- So, the recovery algorithm supports the **no-force policy**: i.e., updated blocks need not be written to disk when transaction commits
  - allows faster commit of transactions
- Similarly, the blocks modified by a transaction that is still active should not be written to disk. This policy is called the **no-steal policy.**
- Alternative, **steal policy**: allows the system to write modified blocks to disk even if the transactions that made those modifications have not all committed.

Database Buffering

T_0: **read** (*A*)

A: - A – 50

Write (*A*)

read (*B*)

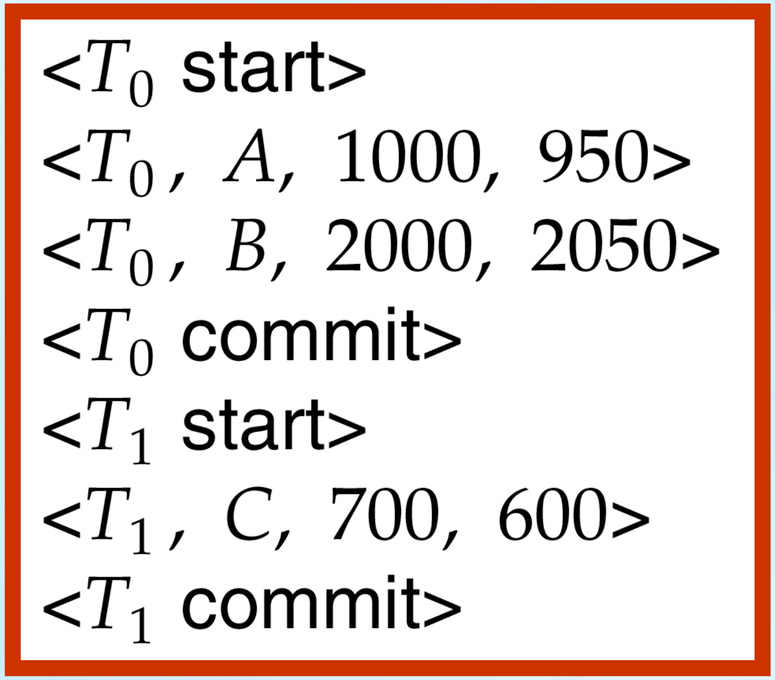
B:- B + 50

write (*B*)

T_1 : **read** (*C*)

C:- C- 100

write (*C*)



<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

Database Buffering

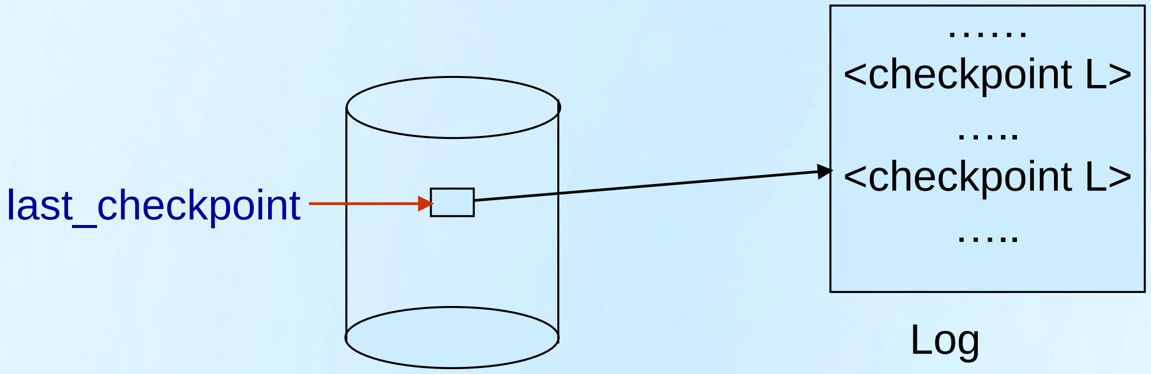
- When a block *B1* is to be output to disk, all log records pertaining to data in *B1* must be output to stable storage before *B1* is output(**WAL**).
- It is important that no writes to the block *B1* be in progress while the block is being output. By using:
- Before a transaction performs a write on a data item, it acquires an exclusive lock on the block in which the data item resides. The lock is released immediately after the update has been performed.
- The following sequence of actions is taken when a block is to be output:
  - Obtain an exclusive lock on the block, to ensure that no other transaction is performing a write on the block.
  - Output log records pertaining to block *B1* to stable storage.
  - Output block *B1 to disk.*
  - Release the lock once the block output has completed.
    - Such locks held for short duration are called **latches**

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
  - If a failure occurs that result in the loss of physical database blocks, the most recent dump is used in restoring the database to a previous consistent state.
  - 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

End of Chapter