Database Systems
Lecture22 – Chapter 19: Recovery System

SUNG KYUN KWAN UNIVERSITY

Beomseok Nam (남범석)

bnam@skku.edu

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 disk failure destroys all or part of disk storage
  - disk drives use checksums to detect failures

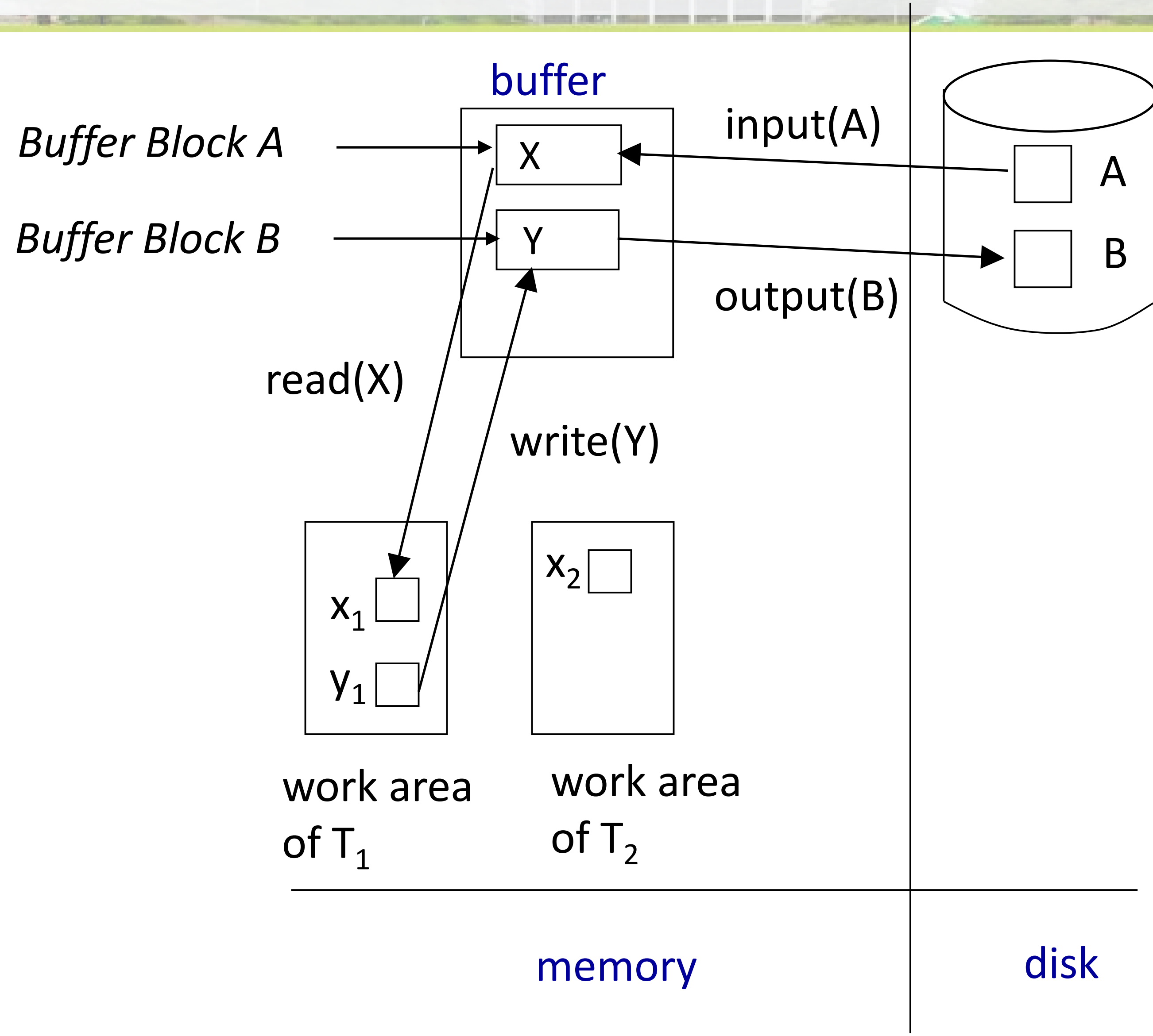
Recovery Algorithms

- Suppose transaction T_i transfers \$50 from account A to account B
  - Two updates:
    - subtract 50 from A
    - add 50 to B
- Transaction T_i requires updates to A and B to be written to DB.
  - A failure may occur before both of modifications are made.
  - Non-atomic modifications result in inconsistent DB
  - 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 a single block.

Example of Data Access

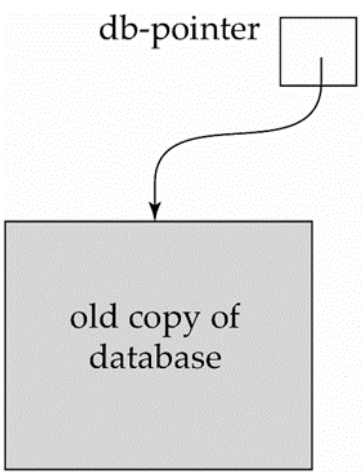


Data Access (Cont.)

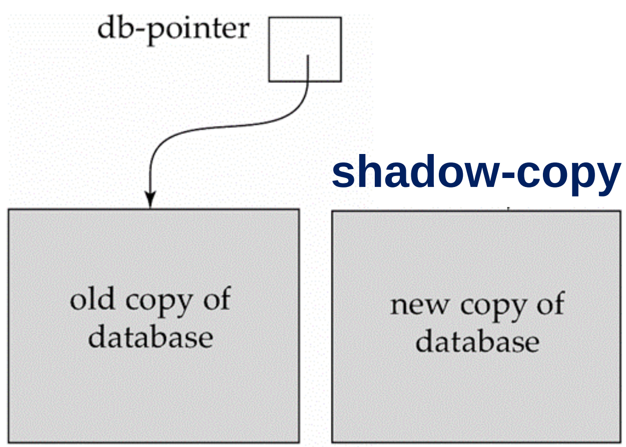
- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - T_i's local copy of a data item X is called x_i.
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read**(X) assigns the value 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

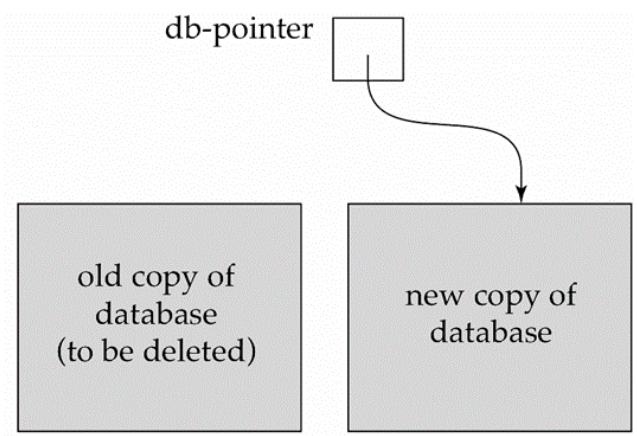
- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study **log-based recovery mechanisms** in detail
  - We first present key concepts
  - And then present the actual recovery algorithm
- Less used alternative: **shadow-copy** and **shadow-paging**



(a) Before update



(a) Before update

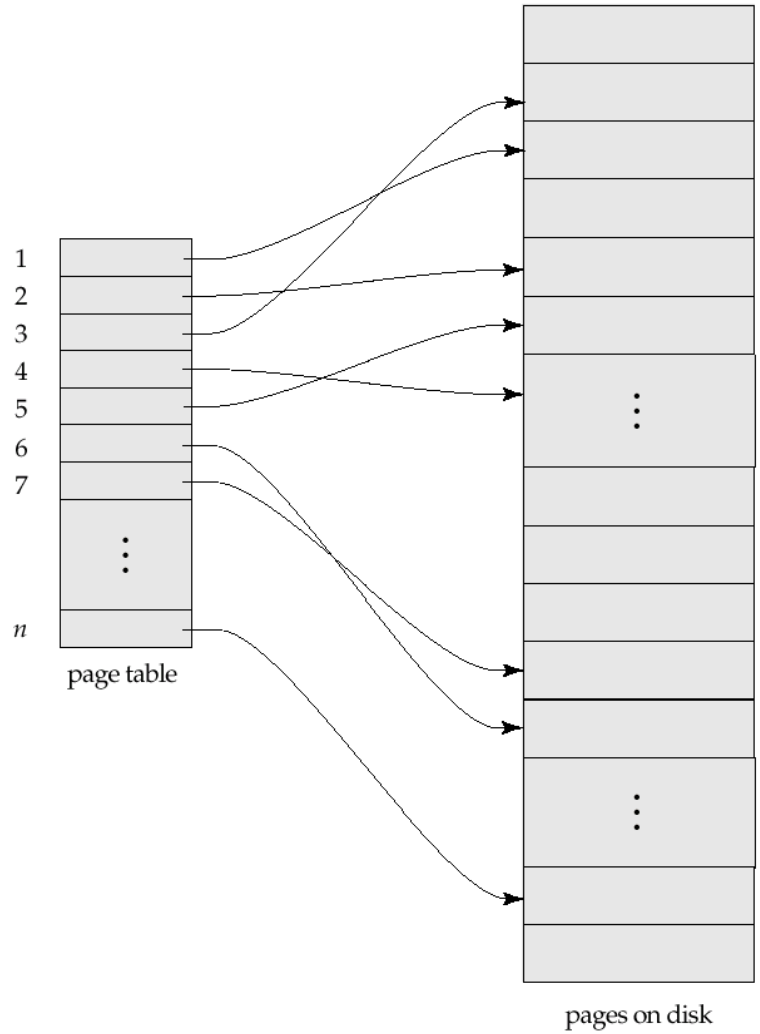


(b) After update

Shadow Paging

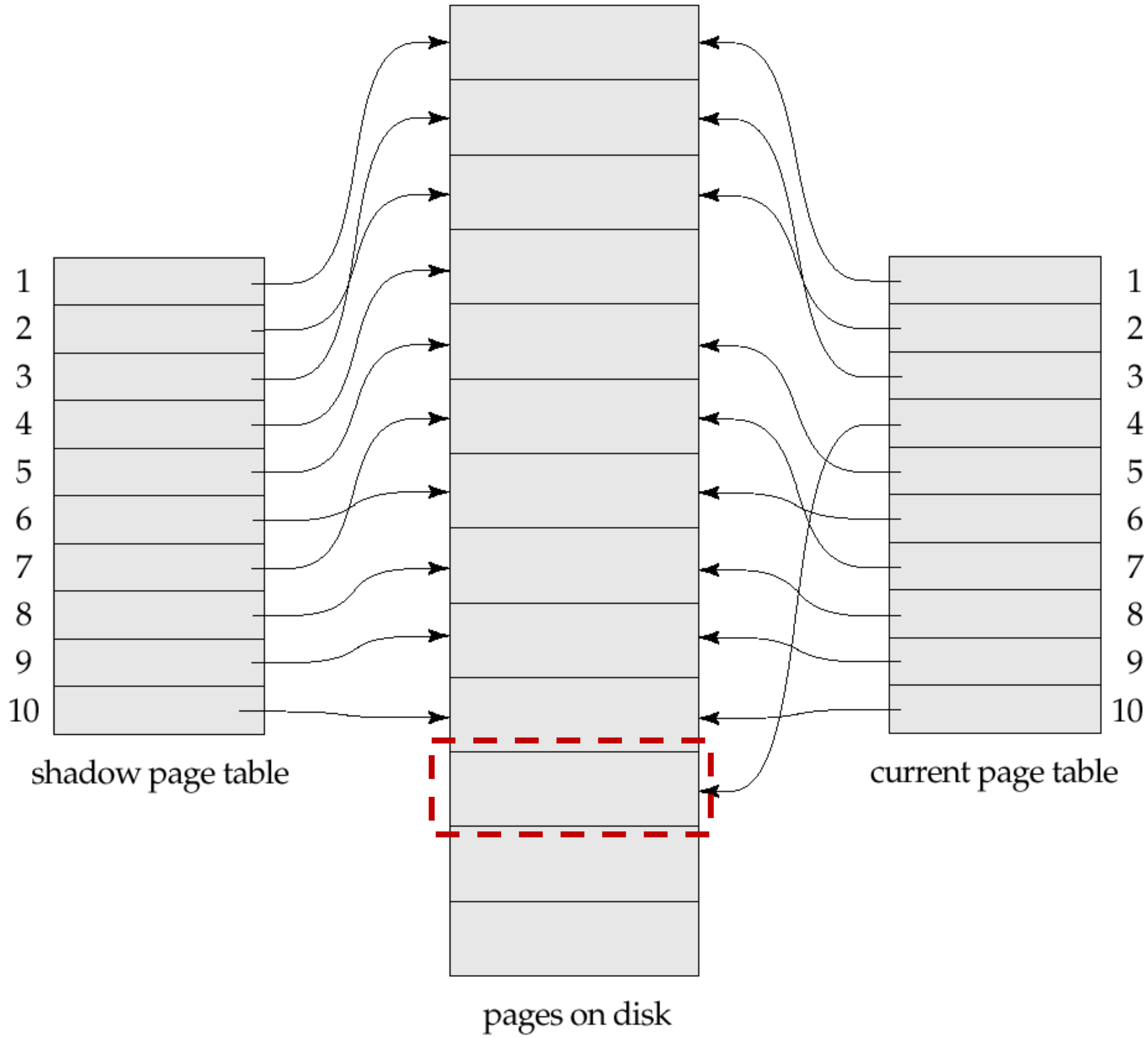
- **Shadow paging** is an alternative to log-based recovery; this scheme is useful if transactions execute serially
- Idea: maintain *two* page tables during the lifetime of a transaction – the **current page table**, and the **shadow page table**
- Store the shadow page table in nonvolatile storage, 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.
- Whenever any page is about to be written for the first time
  - A copy of this page 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

Sample Page Table



Example of Shadow Paging

- 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 is trivial
- Disadvantages :
  - Copying the entire page table is very expensive
  - 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

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

  <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 **old value,** and V_2 is the **new value** to be written to X.
- When T_i finishes its last statement, the log record <T_i **commi**t> 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 to disks *before* database item is written
    - We assume that the log record is output directly to stable storage
  - 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**
<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., 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 <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

Recovering from Failure

- When recovering after failure:
  - Transaction T_i needs to be **undone** if the log
    - Contains <T_i **start**>,
    - But does **not contain** either <T_i **commit**> *or* <T_i **abort**>.
  - Transaction T_i needs to be **redone** if the log
    - Contains <T_i **start**>
    - And **contains** the record <T_i **commit**> *or* <T_i **abort**>

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

Recovering from Failure

- Suppose that transaction T_i was undone earlier and the **<T_i abort>** 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

Checkpoints

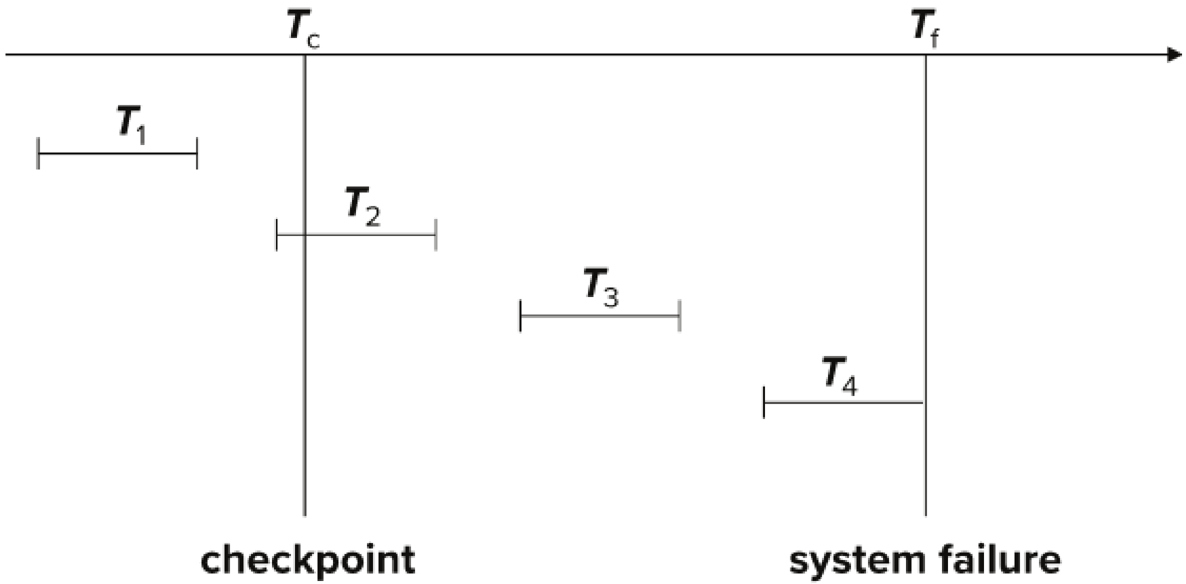
- Redoing/undoing all transactions recorded in the log can be slow
  - Processing the entire log is time-consuming if the system has run for a long time
  - We might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically 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 transactions that didn't finish before the checkpoint, and transactions that started after the checkpoint
  - Scan backwards 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*.

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