Recovery System
Lecture 7

Database System Concepts, 6th Ed.

Recovery System

- Purpose of Database recovery
- Failure types and commit protocols.
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Remote Backup Systems

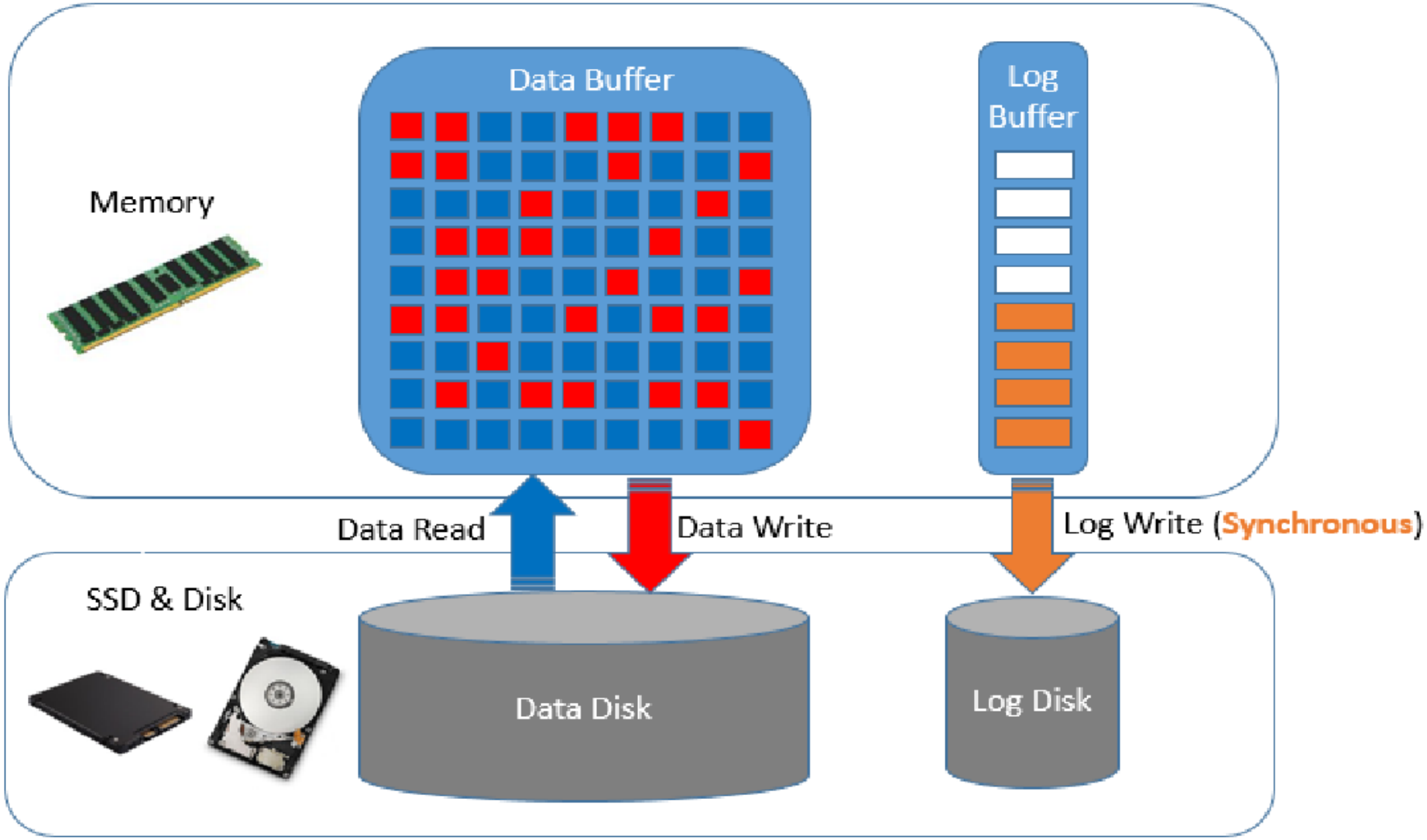
Database Recovery

Purpose of Database Recovery

- To bring the database into the last consistent state, which existed prior to the failure.
- To preserve transaction properties (Atomicity, Consistency, Isolation and Durability).

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


Memory
Data Buffer
Log Buffer
Data Read
Data Write
Log Write (Synchronous)
SSD & Disk
Data Disk
Log Disk

Failure types

1. Soft Failure (system crash)

Causes the loss in volatile memory. Here, the information stored in the non-persistent storage(main memory, buffers, caches or registers) is lost.

The causes of soft failures may be:

- Operating system failure.
- Main memory crash.
- Transaction failure or abortion.
- System generated error like integer overflow or divide-by-zero error.
- Power failure.

2. Hard Failure (Disk failure)

- Causes loss of data in the persistent or non-volatile storage like disk. Disk failure may cause corruption of data in some disk blocks or failure of the total disk.

The causes of a hard failure may be:

  - Power failure.
  - Faults in media.
  - Read-write malfunction عطل.
  - Corruption of information on the disk.
  - Read/write head crash of disk.

- Recovery from disk failures can be short, if there is a new, formatted, and ready-to-use disk on reserve.

3. Network Failure

- It includes the errors induced in the database system due to the distributed nature of the data and transferring data over the network.

The causes of network failure may be−

- Communication link failure.
- Network congestion.
- Site failures.
- Network partitioning.

Commit Protocols

- Are used to ensure the atomicity across sites: a transaction that executed at different sites must be committed or aborted at all sites
- Any database system should guarantee that the desirable properties of a transaction are maintained even after failures.
- If a failure occurs during the execution of a transaction, it may happen that all the changes brought about by the transaction are not committed. This makes the database inconsistent.
- Commit protocols prevent this scenario using either transaction undo (roll-back) or transaction redo (roll-forward).

- Commit Point

The point of time at which the decision is made whether to commit or abort a transaction,

Properties of a commit point:

- It is a point of time when the database is consistent.
- The modifications brought about by the database can be seen by the other transactions. All transactions have a consistent view of the database.
- At this point, all the operations of transaction have been successfully executed and their effects have been recorded in transaction log.
- At this point, a transaction can be safely undone, if required.
- At this point, a transaction releases all the locks held by it.

- Transaction Undo (rollback)

The process of undoing all the changes made to a database by a transaction. This is mostly applied in case of soft failure.

- Transaction Redo (rollforward)

The process of reapplying the changes made to a database by a transaction. This is mostly applied for recovery from a hard failure.

Transaction Log

- A sequential file that keeps track of transaction operations on database items. As it is sequential in nature, it is processed sequentially either from the beginning or from the end.

Purposes of a transaction log −

- To support commit protocols to commit or support transactions.
- To aid database recovery after failure.

- It is usually kept on the disk, so that it is not affected by soft failures. It is periodically backed up to an archival storage like magnetic tape to protect it from disk failures as well.

Transaction Log

For recovery from any type of failure data values prior to modification (BFIM - BeFore Image) and the new value after modification (AFIM – AFter Image) are required. These values and other information is stored in a sequential file called Transaction log. A sample log is given below. **Back P** and **Next P** point to the previous and next log records of the same transaction.

T ID	Back P	Next P	Operation	Data item	BFIM	AFIM
T1	0	1	Begin			
T1	1	4	Write	X	X = 100	X = 200
T2	0	8	Begin			
T1	2	5	W	Y	Y = 50	Y = 100
T1	4	7	R	M	M = 200	M = 200
T3	0	9	R	N	N = 400	N = 400
T1	5	nil	End			

Lists in Transaction Logs

- The transaction log maintains five types of lists depending upon the status of the transaction.
- These lists aid the recovery manager to ascertain the status of a transaction.

Lists are as follows −

- **Commit list :** contains the committed transactions that has a transaction start record and a transaction commit record.
- **Failed list:** contains the failed transactions that has a transaction start record and a transaction failed record but not a transaction abort record.
- **Abort list :** contains the aborted transactions that has a transaction start record and a transaction abort record.
- **Before-commit list**: contains the before-commit transactions that has a transaction start record and a transaction before-commit record (a transaction where all the operations have been executed but not committed)
- **Active list:** contains the active transactions that has a transaction start record but no records of before-commit, commit, abort or failed,

Immediate Update and Deferred Update

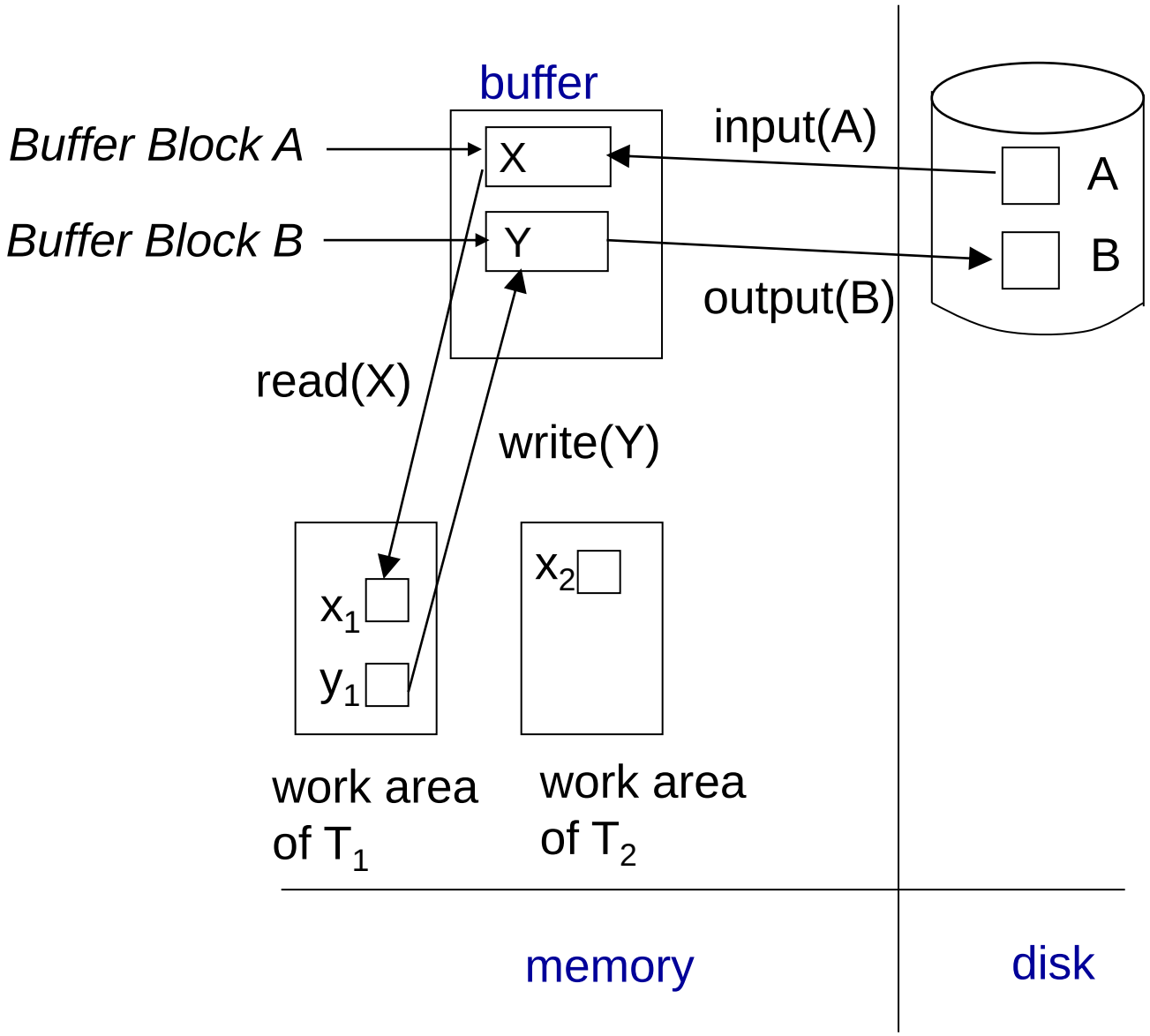
Immediate and Deferred Update are two methods for maintaining transaction logs.

Transaction Status	Immediate update	Deferred update
Executed	Updates written on the disk, The old values and the updates values are written onto the log before writing to the database in disk	Updates recorded on log file
On commit	Changes are made permanent	Changes written on the disk
On rollback	Changes discarded and the old values are restored from old values in the og	Changes on the log discarded and no updates on database

Storage Structure

- **Volatile storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory,
  - but may still fail, losing data
- **Stable storage**:
  - A form of storage that survives all failures
  - Approximated by maintaining multiple copies on distinct nonvolatile media

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 (تراه مناسبا).
- 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

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**$\rangle$ is written.
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

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 T1 updates A, then T2 updates A and commits, and finally T1 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** 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 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

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 greatly

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

Checkpoints

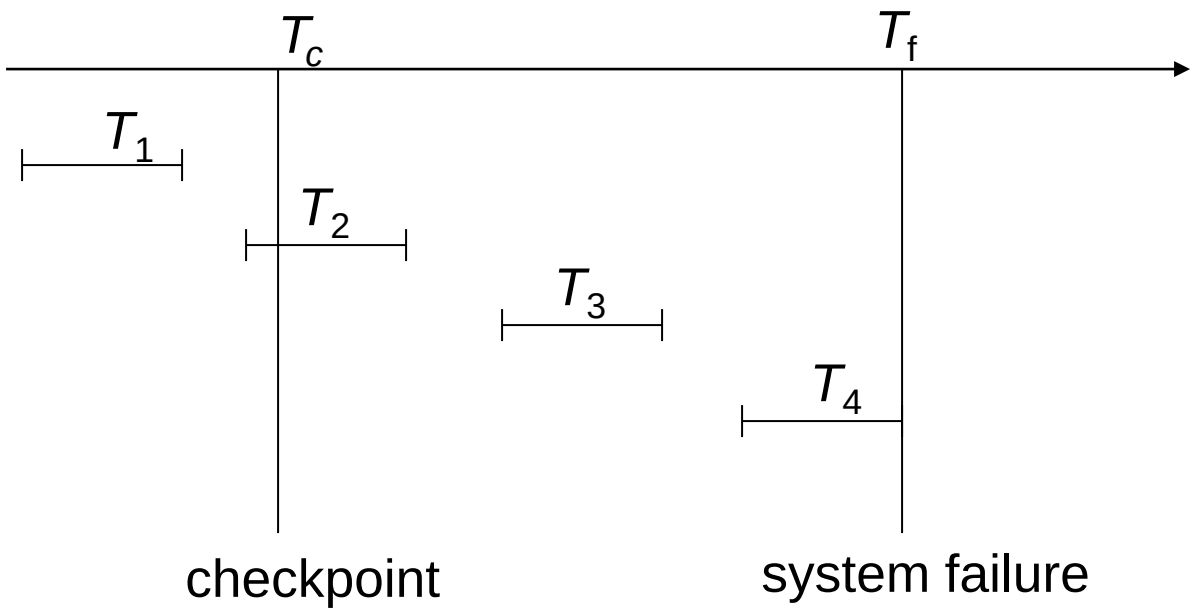
- Redoing/undoing all transactions recorded in the log can be very slow
  1. processing the entire log is time-consuming if the system has run for a long time
  2. we might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically 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 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.
  1. 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
  1. 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