Recovery

- Recovery
  - Catastrophic failure
  - Non-catastrophic failure
- Write-Ahead Logging (WAL) protocol
- Checkpointing
- Deferred Update – NO UNDO/REDO
- Immediate Update – UNDO/REDO

Recovery

- Recovery – *restore* the database to the most *recent consistent state* before the time of failure.
- What is needed to restore? -> information
- Keep information about the changes that were applied to data items by the various transactions -> *system log*
  - Catastrophic failure – disk crash
  - Non-catastrophic failure

Recovery – catastrophic failure

- Restore the past copy from the back-up (archive)
- Reconstruct a more current state by *redoing* the operations of committed transactions from the *backed-up log*, up to the time of failure.

Recovery – non-catastrophic failure

- Identify any changes that may cause an inconsistency in the database
- A transaction that has updated some database items on disk but has *not been committed*
  - Reverse the changes by *undoing* write operations
- A transaction has *committed* but some of its write operations have not yet been written to disk
  - *Redo* the operations to restore a consistent state

Recovery – non-catastrophic failure

- The recovery protocol does not need a complete archival copy of the database
- Rather, the entries kept in the online *system log* on disk are analyzed to determine the appropriate actions for recovery
- Two main policies for recovery: deferred update and immediate update

Write-Ahead Log (WAL) protocol

- The two types of log entry information included for a write command: the information needed for 1) UNDO 2) REDO
- A REDO-type log entry includes the *new value* (AFIM) of the item written by the operation
- REDO-type log entry: [`write_item, T, X, new_value`]
- The UNDO-type log entries include the *old value* (BFIM) of the item
- UNDO-type: [`write_item, T , X , old_value, new_value`]

Write-Ahead Log (WAL) protocol

		A	B	C	D
		30	15	40	20
	[start_transaction,T_3]				
	[read_item,T_3,C]				
*	[write_item,T_3,B,15,12]		12		
	[start_transaction,T_2]				
	[read_item,T_2,B]				
**	[write_item,T_2,B,12,18]		18		
	[start_transaction,T_1]				
	[read_item,T_1,A]				
	[read_item,T_1,D]				
	[write_item,T_1,D,20,25]				25
	[read_item,T_2,D]				
**	[write_item,T_2,D,25,26]				26
	[read_item,T_3,A]				

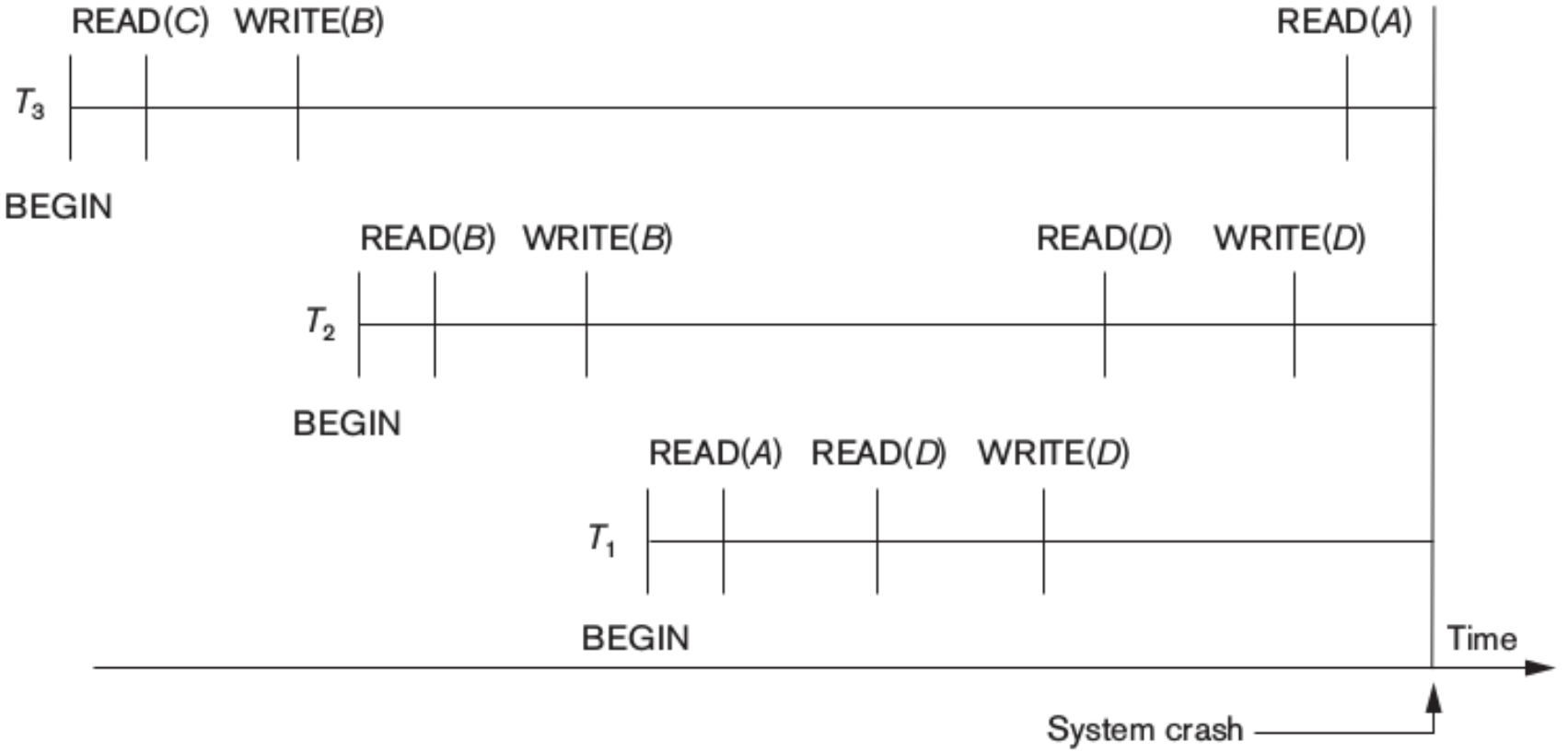
← System crash

point of crash. (c) Op
before the crash.

* T_3 is rolled back because it did not reach its commit point.

** T_2 is rolled back because it reads the value of item B written by T_3.

Write-Ahead Log (WAL) protocol



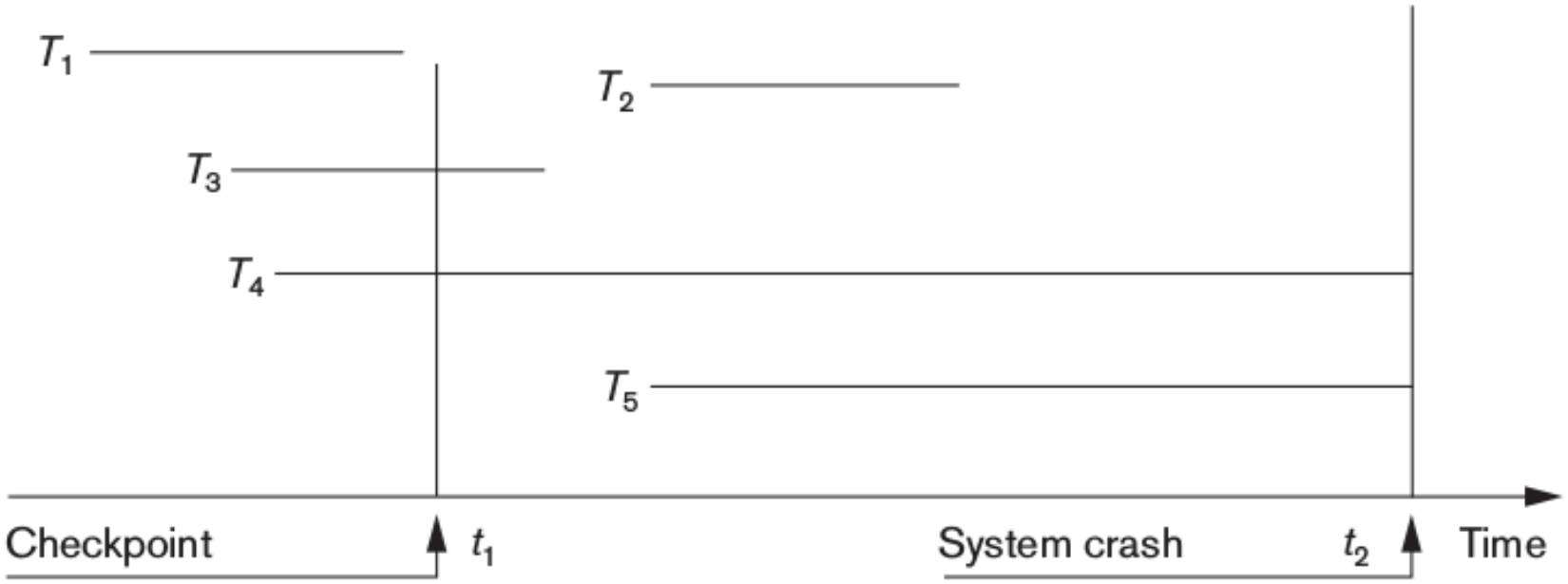
Write-Ahead Log (WAL) protocol

- To permit recovery, the appropriate entries must be permanently recorded in the log on disk *before changes are applied to the database*
- Before the BFIM is overwritten with the AFIM in the database, ensure that
- a) the BFIM of the data item is recorded in the appropriate log entry
- b) the log entry is flushed to disk

Write-Ahead Log (WAL) protocol

- WAL protocol that requires both UNDO and REDO:
- The BFIM of an item cannot be overwritten by its AFIM in the database on disk until all UNDO-type log entries have been force written to disk
- The commit operation of a transaction cannot be completed until all the REDO-type and UNDO-type log records for that transaction have been force written to disk

Checkpointing



Checkpointing

- Another type of entry in the log is called a checkpoint
- A [checkpoint, list of active transactions] record is written into the log periodically
- All transactions that have their [commit, T] entries in the log before a [checkpoint] entry *do not need* to have their WRITE operations *redone* in case of a system crash
- Since their updates will be recorded in the database on disk during checkpointing

Checkpointing

- The list of transaction ids for *active transactions* at the time of the checkpoint is included in the checkpoint record
- These active transactions can be easily identified during recovery
- The recovery manager must decide at what intervals to take a checkpoint

Checkpointing

- A checkpoint consists of the following actions:
- 1. Suspend execution of transactions temporarily.
- 2. Force-write all main memory buffers that have been modified to disk.
- 3. Write a [checkpoint] record to the log, and force-write the log to disk.
- 4. Resume executing transactions.

NO-UNDO/REDO Recovery Based on Deferred Update

- Idea – to defer or postpone any actual updates to the database on disk until the transaction completes its execution successfully and reaches its commit point.
- Before the commit (during the transaction):
  - The updates are recorded only in the log and in the cache buffers that DBMS maintains
  - Does not physically update the database on disk

NO-UNDO/REDO Recovery Based on Deferred Update

- After the transaction reaches its commit point:
  - the log is force-written to disk – the updates are recorded persistently in the log file on disk
  - the updates are written to the database from the main memory buffers

NO-UNDO/REDO Recovery Based on Deferred Update

- A typical deferred update protocol stated as follows:
- A transaction cannot change the database on disk until it reaches its commit point
- A transaction does not reach its commit point until all its REDO-type log entries are recorded in the log and the log buffer is force-written to disk (WAL)

NO-UNDO/REDO Recovery Based on Deferred Update

- During recovery: only REDO-type log entries are required
- Why? - the system fails after a transaction commits but before all its changes are recorded in the database on disk
- UNDO-type log entries are not needed, because the transaction has not affected the database on disk

NO-UNDO/REDO Recovery Based on Deferred Update

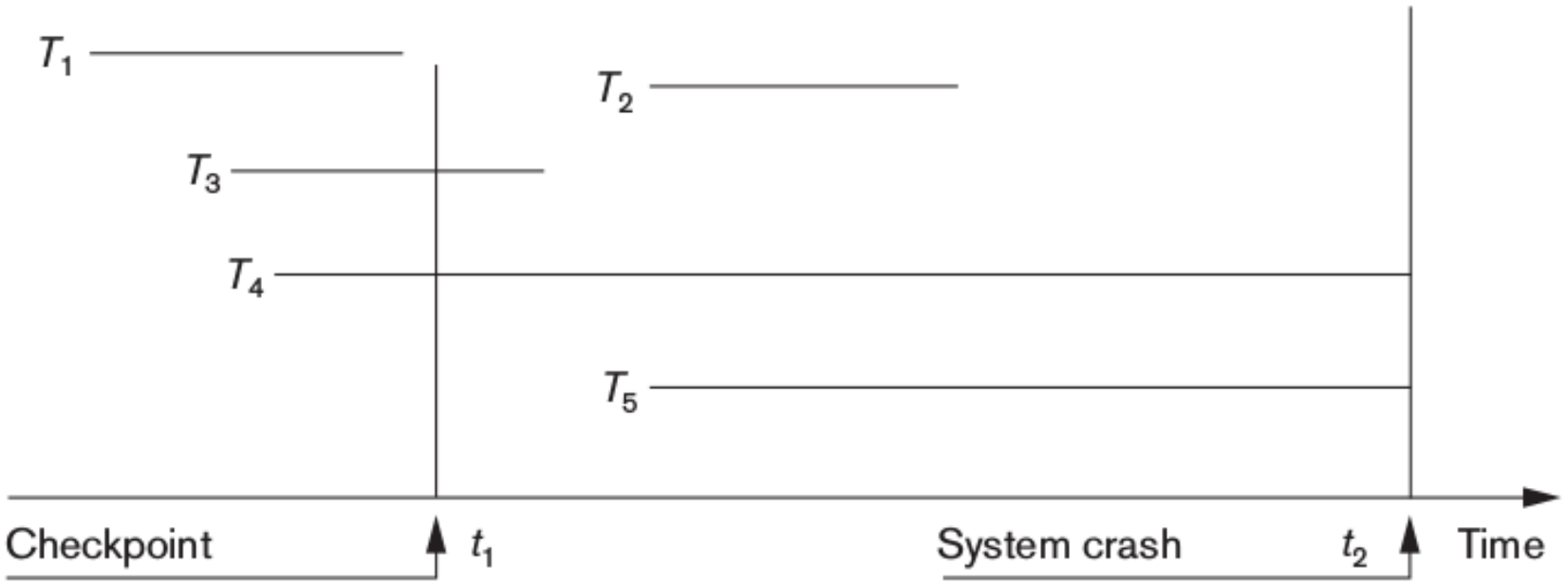
- Use two lists of transactions:
- The committed transactions T since the *last checkpoint* (commit list), and the active transactions T′ (active list)
- REDO all the WRITE operations of the committed transactions from the log, *in the order in which they were written into the log*
- The transactions that are active and did not commit are effectively canceled and must be resubmitted

NO-UNDO/REDO Recovery Based on Deferred Update

- REDO Procedure:
- Redoing a write_item operation is check its log entry [`write_item, T, X, new_value`] and set the value of item X in the database to `new_value` – AFIM.

NO-UNDO/REDO Recovery Based on Deferred Update

- Commit list: T2, T3 – REDO
- Active list: T4, T5 – cancelled / resubmitted



NO-UNDO/REDO Recovery Based on Deferred Update

- When concurrent execution is permitted, the *recovery process again depends on* the concurrency control protocol
- Consider the recovery along with concurrency control for a multiuser system
- Assuming strict 2PL – write (exclusive) locks are not released until the transaction terminates

T_1
read_item(A)
read_item(D)
write_item(D)

T_2
read_item(B)
write_item(B)
read_item(D)
write_item(D)

T_3
read_item(A)
write_item(A)
read_item(C)
write_item(C)

T_4
read_item(B)
write_item(B)
read_item(A)
write_item(A)

[start_transaction,T_1]
[write_item, T_1, D, 20]
[commit, T_1]
[checkpoint]
[start_transaction, T_4]
[write_item, T_4, B, 15]
[write_item, T_4, A, 20]
[commit, T_4]
[start_transaction, T_2]
[write_item, T_2, B, 12]
[start_transaction, T_3]
[write_item, T_3, A, 30]
[write_item,T_2, D, 25] ← System crash

T_2 and T_3 are ignored because they did not reach their commit points.

T_4 is redone because its commit point is after the last system checkpoint.

Figure 22.3
An example of recovery using deferred update with concurrent transactions. (a) The READ and WRITE operations of four transactions. (b) System log at the point of crash.

Deferred Update Advantages

- A transaction does not record any changes in the database on disk until after it reaches commit – hence it is never rolledback!
- No dirty read – Hence, no cascading rollback will occur.

Deferred Update Limitations

- Require excessive buffer space to hold all updated items until the transactions commit
- Cannot be used in practice unless transactions are short and each transaction changes few items

UNDO/REDO Recovery Based on Immediate Update

- When a transaction issues an update command, the database on disk can be updated immediately
- No need to wait for the transaction to reach its commit point
- If the transaction is allowed to commit before all its changes are written to the database – UNDO/REDO recovery algorithm

UNDO/REDO Recovery Based on Immediate Update

- 1. Use two lists of transactions: the committed transactions and the active transactions.
- 2. Undo all the write_item operations of the active transactions – UNDO. Undo in the *reverse of the order* in which they were written into the log.
- 3. Redo all the write_item operations of the committed transactions from the log, in the *order in which they were written* into the log – REDO

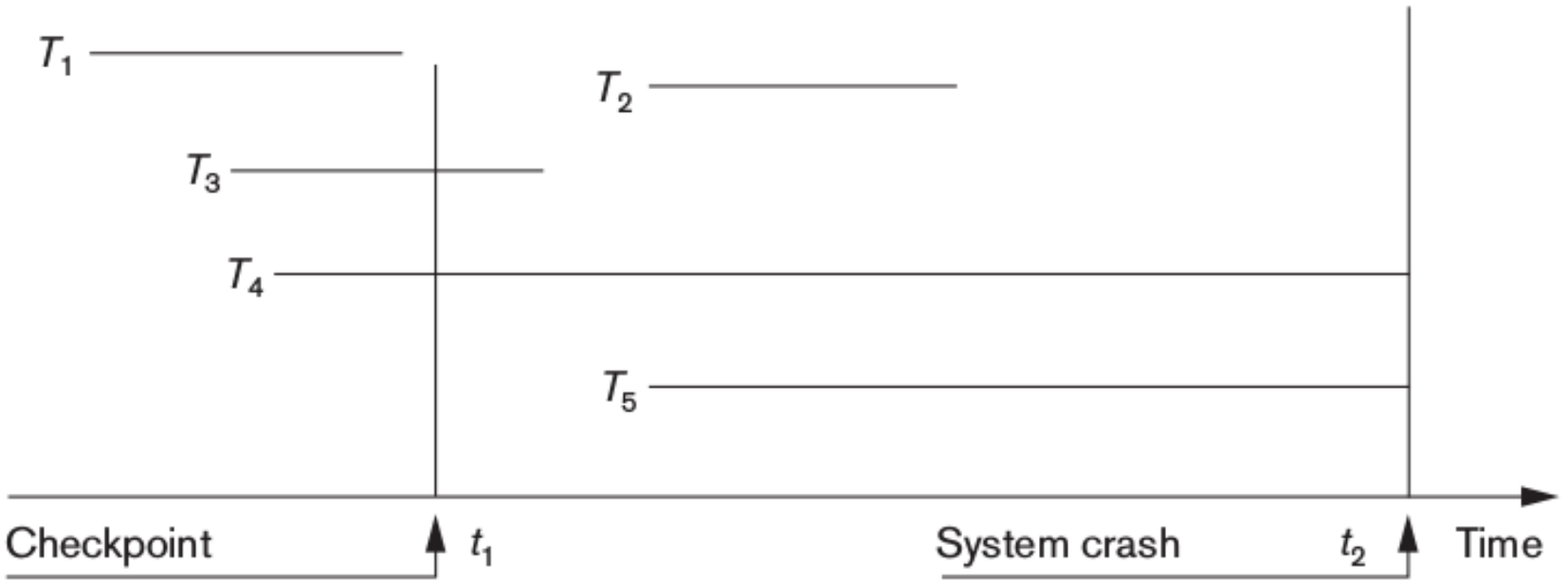
UNDO/REDO Recovery Based on Immediate Update

- Procedure UNDO:
- Examine its log entry

  [`write_item, T , X , old_value, new_value`] and set the value of item X in the database to `old_value` – BFIM
- Undoing a write_item operations from the log must proceed in the *reverse order from the order in which the operations were written in the log*

UNDO/REDO Recovery Based on Immediate Update

- Commit list: T2, T3 – REDO
- Active list: T4, T5 – UNDO



UNDO/REDO Recovery Based on Immediate Update

- [start_transaction, T1]
  [write_item, T1, D, 20, 25]
  [checkpoint, tc]
  [start_transaction, T2]
  [write_item, T2, B, 12, 18]
  [commit, T1]
  [start_transaction, T3]
  [write_item, T3, D, 25, 15]
  [start_transaction, T4]
  [write_item, T4, C, 30, 40]
  [write_item, T3, A, 30, 20]
  [commit, T3]
  [write_item, T2, D, 15, 25]

<-------------- system crash

- Assume strict 2PL:
- What is the output?
- Which are REDONE and UNDONE?