Database Technology

Topic 11: Database Recovery

Olaf Hartig

olaf.hartig@liu.se

LINKÖPING UNIVERSITY

Types of Failures

Database may become unavailable for use due to:

- Transaction failures
  - e.g., incorrect input, deadlock, incorrect synchronization
  - Result: transaction abort
- System failures
  - e.g., application error, operating system fault
- Media failures
  - e.g., RAM failure, disk head crash, power disruption

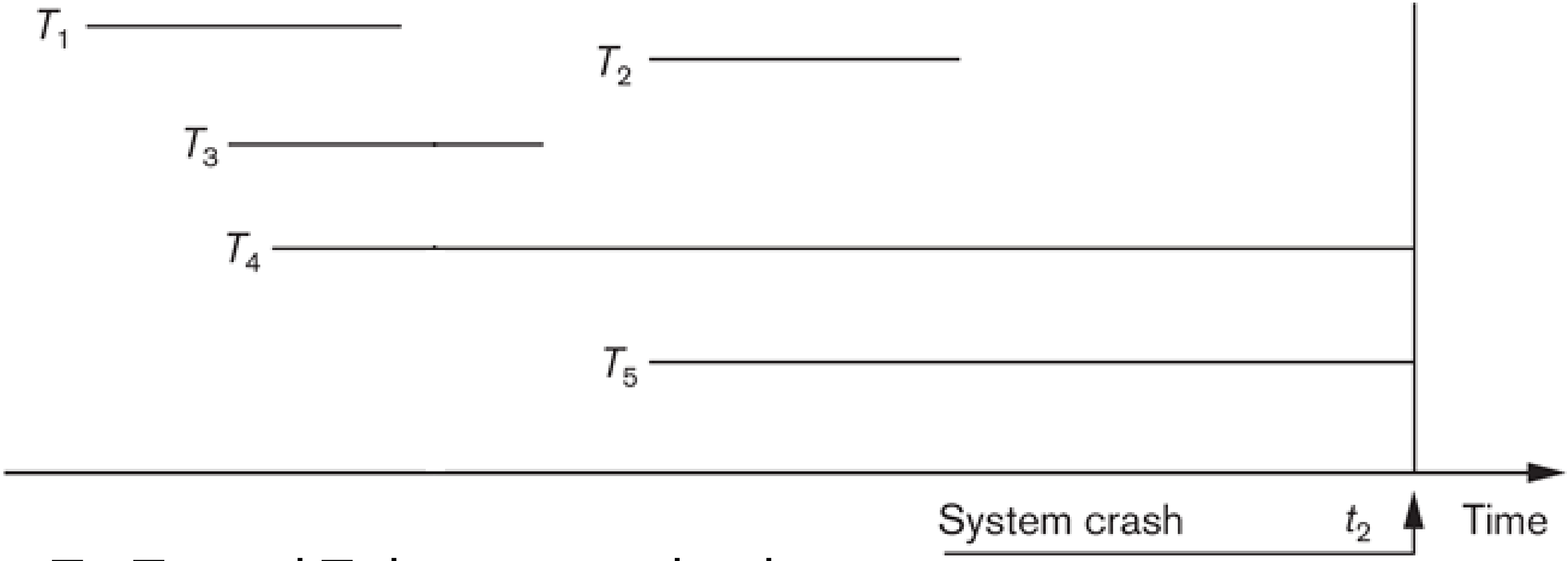
Focus of the rest of the lecture:

- We wish to recover from system failures
  - Recovery from media failures similar, but may need to restore database files from *backup*

Situation after System Failure

- DBMS is halted abruptly
- Processing of in-progress SQL commands halted abruptly
- Connections to application programs (clients) are broken
- States of executing programs unknown
- Contents of memory buffers are lost
- Database files are *not* damaged

Problem Situation Example



- T_1, T_2, and T_3 have committed
- T_4 and T_5 still in progress
- Any of the transactions might have written data
- Some (unknown) subset of the writes have been flushed to disk

Purpose of Database Recovery

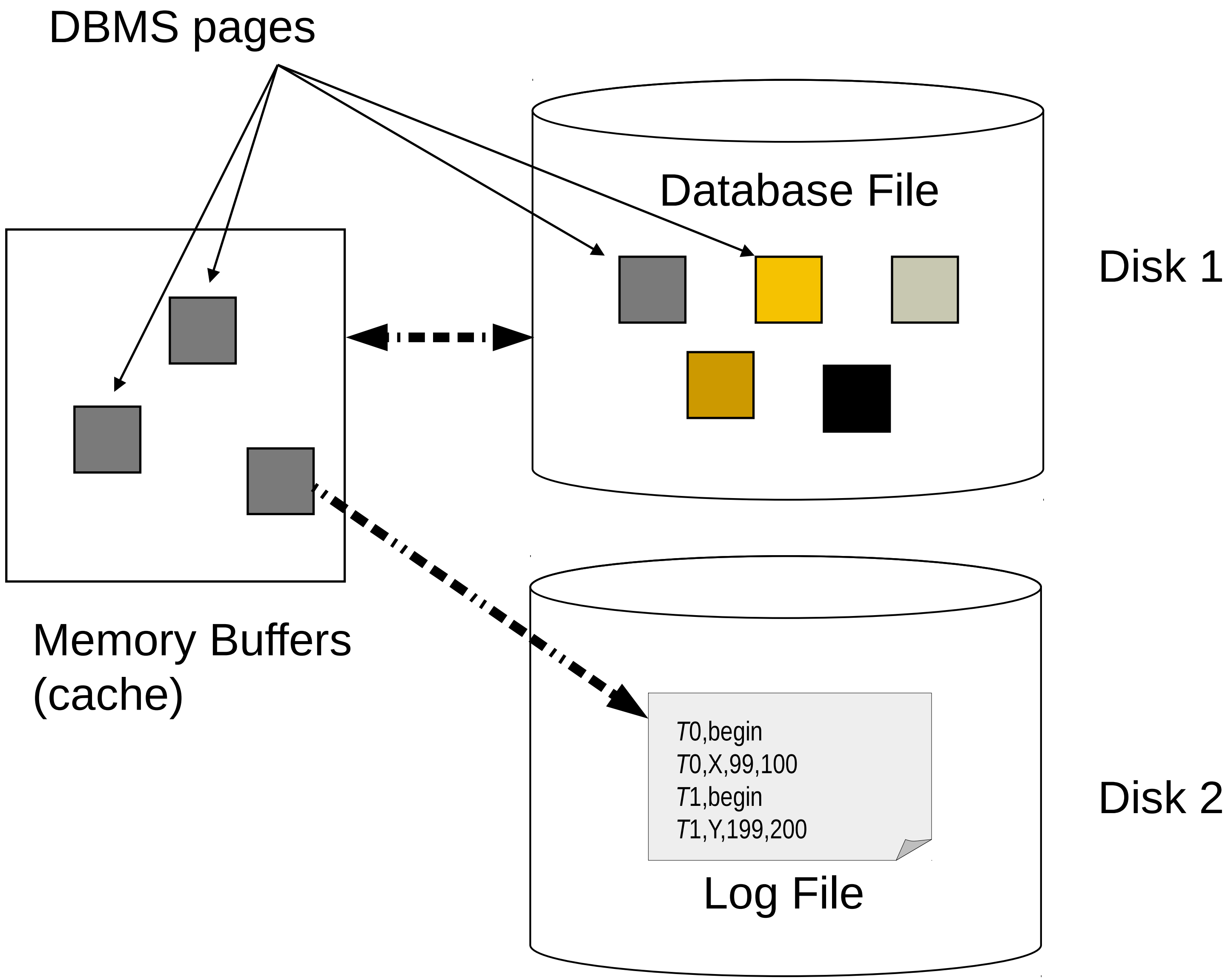
- Bring the database into the most recent consistent state that existed prior to a failure
- Atomicity and Durability of the ACID properties
  - Abort (and restart) TAs active at time of failure
  - Ensure changes made by committed TAs are not lost
- Complication due to database execution model:
  - Data items packed into I/O blocks (pages)
  - At time of write updated data first stored in main memory buffer
  - Actually written to disk some time later

Logging

System Log

- Append-only file
  - Keep track of all operations of all transactions
  - In the order in which operations occurred
- Stored on disk
  - Persistent except for disk or catastrophic failures
  - Periodically backed up (to guard against disk and catastrophic failures)
- Log buffer in main memory
  - Holds log records that are appended to the log
  - Occasionally whole buffer appended to end of log file on disk (flush)

Storage Structure



```
T0,begin
T0,X,99,100
T1,begin
T1,Y,199,200
```

Log Records

- [**start_transaction**, *T*]
  - Transaction *T* has started execution
- [**write_item**, *T*, *X*, *old_value*, *new_value*]
  - Transaction *T* has changed the value of item *X* from *old_value* to *new_value*
  - *old_value* (before image) needed to **undo(*X*)**
  - *new_value* (after image) needed to **redo(*X*)**
- [**commit**, *T*]
  - *T* has completed successfully and committed
  - Effects (writes) of *T* must be durable
- [**abort**, *T*]
  - *T* has been aborted
  - Effects (writes) of *T* must be ignored and undone

Commit Point

- A transaction reaches its commit point when:
  1. all of its operations are executed, and
  2. all its log records are flushed to disk
     (where the last is the commit record)

- Beyond its commit point
  - the transaction is said to be *committed*, and
  - its effect must be permanently recorded in the DB

Write-Ahead Logging (WAL)

- Used to ensure that the log
  - is consistent with the DB, and
  - can be used to recover the DB to a consistent state
- Two rules:
  1. Log record(s) for a page must be written before corresponding page is flushed to disk, and
  2. All log records must be written before commit
- Rule 1 for atomicity
  - each operation is known and can be undone if needed
- Rule 2 for durability
  - the effect of a committed transaction is known

Recovery Process

Recovery with Deferred Update

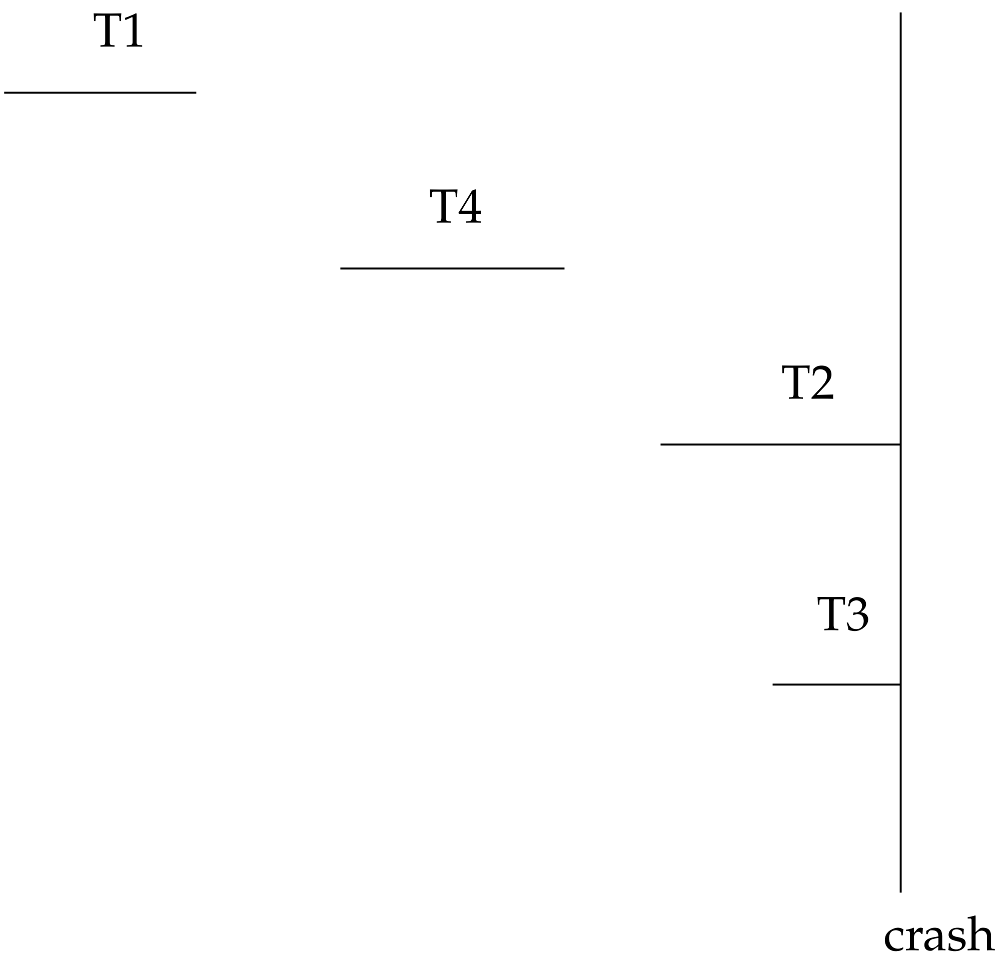
- Updating the DB on disk after each change is inefficient
- **Deferred update:**
  - Updates of a transaction *T* are written to disk *after (but not necessarily immediately after) T* has reached commit point
- No need to undo changes of non-committed transactions
- Need to redo the changes of committed transactions
- **NO-UNDO/REDO recovery algorithm:**
  - Create a list of active (i.e., non-committed) transactions and a list of committed transactions
  - REDO all the write-item operations of all the TAs in the second list in the order in which they appear in the log (use *after image* from the log records)

Example

NO-UNDO

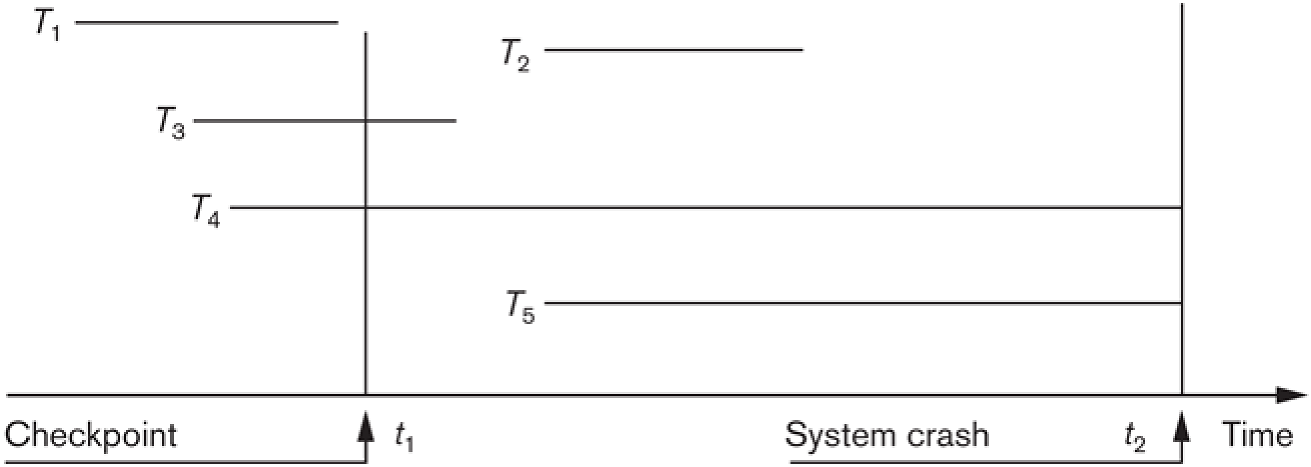
REDO: T1, T4

start-transaction T1
write-item T1, D, 10, 20
commit T1
start-transaction T4
write-item T4, B, 10, 20
write-item T4, A, 5, 10
commit T4
start-transaction T2
~~write-item T2, B, 20, 15~~ *ignore*
start-transaction T3
~~write-item T3, A, 10, 30~~ *ignore*
~~write-item T2, D, 20, 25~~ *ignore*
CRASH



Checkpointing

- To save redo effort, use **checkpoints**
- Occasionally flush data buffers using the five steps:
  1. Suspend execution of transactions temporarily
  2. Force-write modified buffer data to disk
  3. Append [checkpoint] record to log
  4. Flush log to disk
  5. Resume normal transaction execution
- During recovery, redo required only for log records appearing after [checkpoint] record

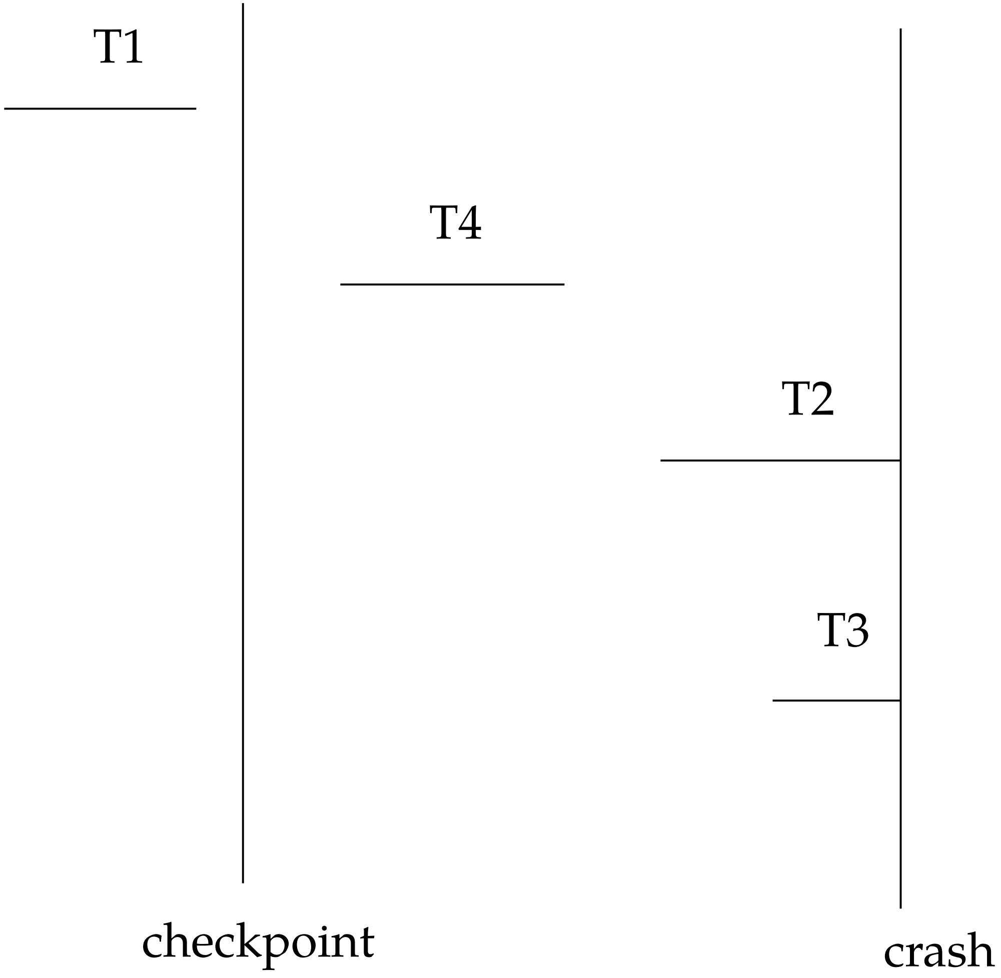


Example with Checkpoint

NO-UNDO

REDO: ~~T1~~, T4

start-transaction T1
~~write-item T1, D, 10, 20~~ *ignore*
commit T1
checkpoint
start-transaction T4
write-item T4, B, 10, 20
write-item T4, A, 5, 10
commit T4
start-transaction T2
~~write-item T2, B, 20, 15~~ *ignore*
start-transaction T3
~~write-item T3, A, 10, 30~~ *ignore*
~~write-item T2, D, 20, 25~~ *ignore*
CRASH



Recovery with Immediate Update 1

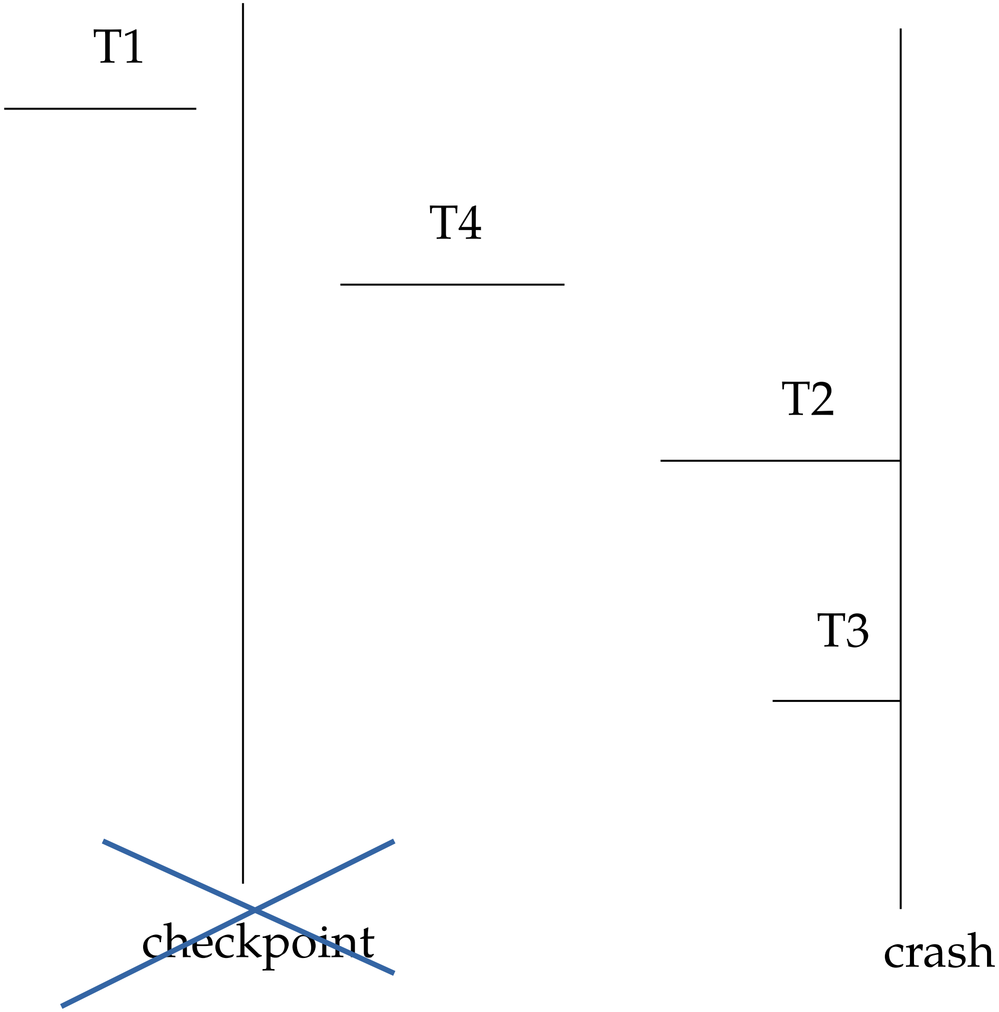
- **Immediate update:**
  - Updates of a transaction may be written to disk *before* the transaction commits (with the log records for such updates being written out first, i.e., write-ahead logging)
- Additional requirement: all updates of a transaction *T* must be written to disk before the commit point of *T*
  - No need to redo changes of committed transactions
  - Need to undo changes of non-committed transactions
- **UNDO/NO-REDO recovery algorithm:**
  - Create a list of active (i.e., non-committed) transactions
  - UNDO all the write-item operations of all the TAs in the list in the *reverse* order in which they appear in the log (use *before image* from the log records)

Example

UNDO: T2, T3

NO REDO

start-transaction T1
~~write-item T1, D, 10, 20~~ *ignore*
commit T1
~~checkpoint~~ *not needed*
start-transaction T4
~~write-item T4, B, 10, 20~~ *ignore*
~~write-item T4, A, 5, 10~~ *ignore*
commit T4
start-transaction T2
write-item T2, B, 20, 15
start-transaction T3
write-item T3, A, 10, 30
write-item T2, D, 20, 25
CRASH



Recovery with Immediate Update 2

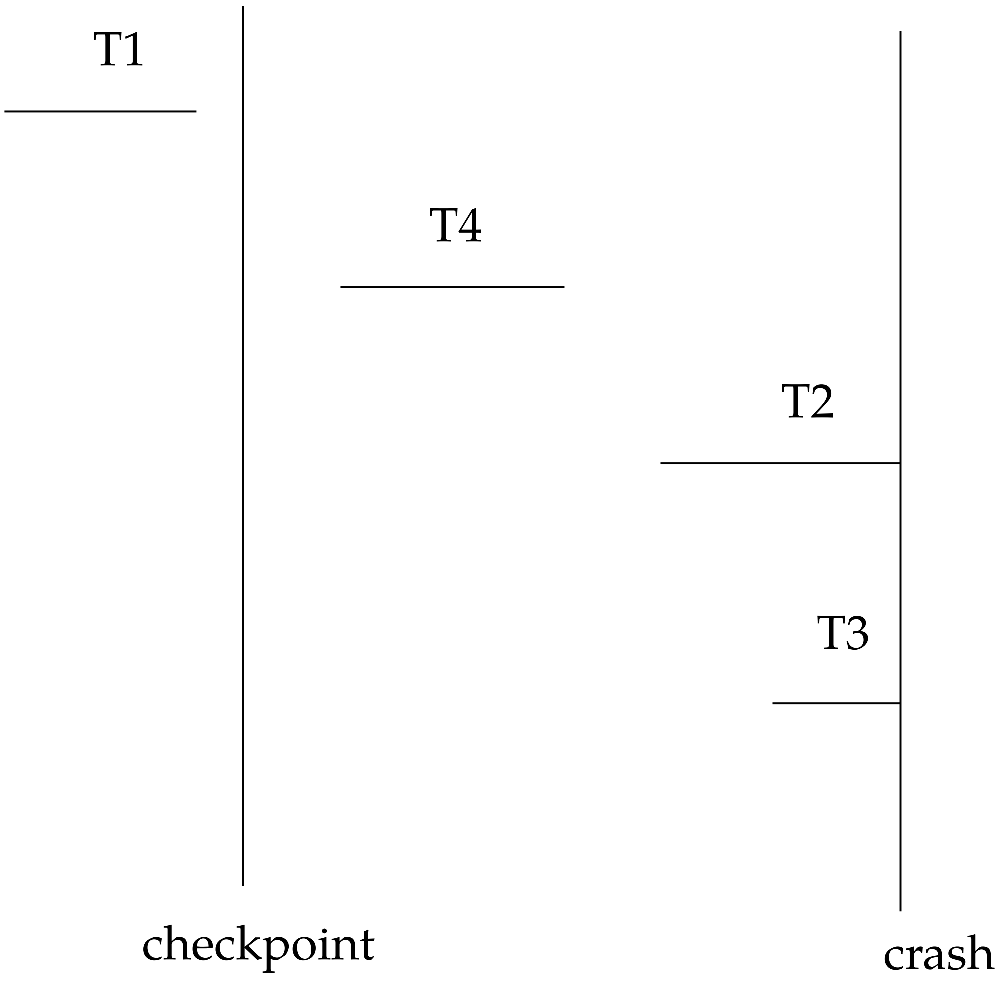
- No additional requirements
- Then:
  - Need to redo changes of committed transactions
  - Need to undo changes of non-committed transactions
- **UNDO/REDO recovery algorithm:**
  - Create a list of active (i.e., non-committed) transactions and a list of committed transactions *since last checkpoint*
  - UNDO all the write-item operations of all the TAs in the first list in the *reverse* order in which they appear in the log (use *before image* from the log records)
  - REDO all the write-item operations of all the TAs in the second list in the order in which they appear in the log (use *after image* from the log records)

Example

UNDO: T2, T3

REDO: T4

start-transaction T1
~~write-item T1, D, 10, 20~~ *ignore*
commit T1
checkpoint
start-transaction T4
write-item T4, B, 10, 20
write-item T4, A, 5, 10
commit T4
start-transaction T2
write-item T2, B, 20, 15
start-transaction T3
write-item T3, A, 10, 30
write-item T2, D, 20, 25
CRASH



Quiz

Which of the following log records include operations that must be *undone* in case of a crash?

Log Seq #	TID	Op	Item	Before Image	After Image
1	T1	Begin			
2	T1	Write	X	100	200
3	T2	Begin			
4	T2	Write	Y	50	100
5	T3	Begin			
6	T1	End			
7	T1	Commit			
8	T3	Write	Y	100	300

A: all of them B: none of them C: 2, 4, 8 D: 3, 4, 5, 8 E: 4, 8

Summary

Summary

- Transaction log
- Transaction roll-back (undo) and roll-forward (redo)
- Checkpointing

www.liu.se

LINKÖPING
UNIVERSITY