Advanced DB

CHAPTER 17
RECOVERY SYSTEM

Chapter 17: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Shadow Paging
- Recovery With Concurrent Transactions
- Buffer Management
- Failure with Loss of Nonvolatile Storage
- Advanced Recovery Techniques
- ARIES Recovery Algorithm
- Remote Backup Systems

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 have been corrupted by system crash
        - Database systems have numerous integrity checks to prevent corruption of disk data
- Disk failure
    - a head crash or similar disk failure destroys all or part of disk storage
    - Destruction is assumed to be detectable: disk drives use checksums to detect failures

Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
  - Focus of this chapter
- 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

Storage Structure

- Volatile storage:
  - does not survive system crashes
  - examples: main memory, cache memory
- Nonvolatile storage:
  - survives system crashes
  - examples: disk, tape, flash memory,
    non-volatile (battery backed up) RAM
- Stable storage:
  - a theoretical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media

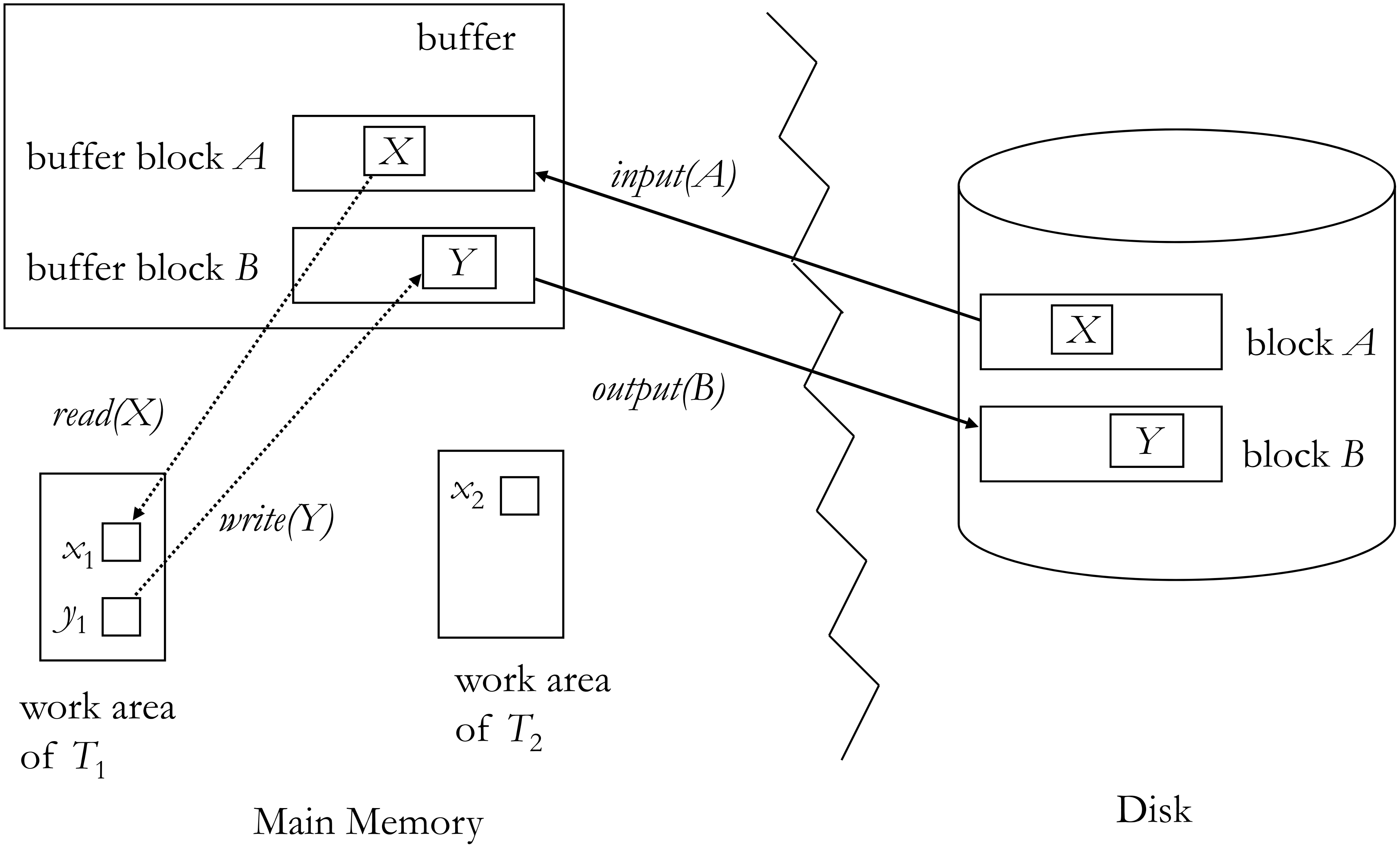
Data Access

- Data blocks
  - Physical blocks: blocks residing on the disk
  - Buffer blocks: blocks residing temporarily in main memory (disk buffer).
- Each transaction T_i has its private work-area
  - local copies of all data items accessed and updated by it are kept here
  - T_i's local copy of a data item X is called x_i.
- Block movements between disk and main memory:
  - **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 that no data item spans two or more blocks.

Data Access (Cont.)

- Transaction transfers data items between system buffer blocks and its private work-area using the following operations :
  - **read**(X)
    - If B_X in which X resides is not in memory, issue **input**(B_X)
    - assign to the local variable x_i the value of X from the buffer block
  - **write**(X)
    - If B_X in which X resides is not in memory, issue **input**(B_X)
    - assign the value x_i to X in the buffer block.
- Transactions
  - perform **read**(X) when accessing X for the first time;
  - All subsequent accesses are to the local copy x_i.
  - After last access, transaction executes **write**(X) if updated.
- output(B_X) need not immediately follow write(X)
  - System can perform the **output** operation when it deems fit.

Example of Data Access



Recovery and Atomicity

- Several output operations may be required for T_i
  - A failure may occur after one of these modifications have been made but before all of them are made
  - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
- To ensure atomicity despite failures
  - we first output information describing the modifications to stable storage without modifying the database itself
  - *Log-based recovery*
  - *Shadow-paging* (not covered in this class)

Log-Based Recovery

- Log
  - a sequence of *log records* that describe update activities on the database
  - Log is kept on stable storage
- Log records
  - When transaction T_i starts: $<T_i$ **start**$>$
  - *Before* T_i executes write(X): $<T_i, X, V_1, V_2>$

    V_1: the value of X before the write

    V_2: the value to be written to X.
  - When T_i finishes its last statement: $<T_i$ **commit**$>$
- Assume that
  - transactions execute serially
  - log records are written directly to stable storage (they are not buffered)

Deferred Database Modification

- Record all modifications to the log, but defer all the *writes* to after *partial commit*
- Logging for Deferred DB Modification
  1. Transaction start: <T_i *start*>
  2. A write(***X***) operation results in
     - <T_i, *X*, *V*> being written to log, where *V* is the new value for *X* (old value is not needed for this scheme)
     - The write is not performed on *X* at this time, but is deferred.
  3. When T_i partially commits
     - <T_i **commit**> is written to the log (and T_i commits)
  4. Finally, the log records are read and used to actually execute the previously deferred writes.

Deferred Database Modification (Cont.)

- Recovery after a crash
  - a transaction needs to be redone if and only if

    both <T_i **start**> and<T_i **commit**> are in the log.
  - **redo** T_i (redoing a transaction T_i) sets the value of all data items updated by the transaction to the new values.
- 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)

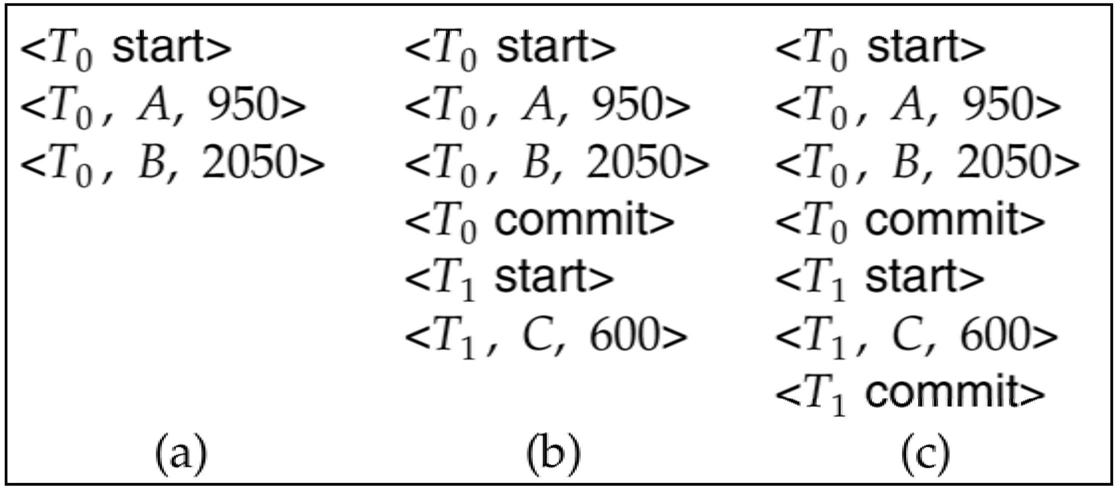
T_1 : read(C)

C :- C-100

write(C)

Deferred Database Modification (Cont.)

- 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), since <T_0 commit> is present
  - (c) redo(T_0) and redo(T_1) since <T_0 commit> and <T_i commit> are present
- Crashes can also occur while recovery action is being taken

Immediate Database Modification

- Allows database updates of an uncommitted transaction to be made as the writes are issued
- Logging for Immediate DB Modification
  1. Transaction start: <T_i *start*>
  2. A **write**(*X*) operation results in
     a. <T_i, X, V_1, V_2> being written to log (undoing may be needed)
     b. followed by the write operation
  3. When T_i partially commits, <T_i **commit**> is written to the log
- 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

Immediate Database Modification (Cont.)

Log	Write	Output
<T_0 start>		
<T_0, A, 1000, 950>		
<T_o, 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_X denotes block containing X)

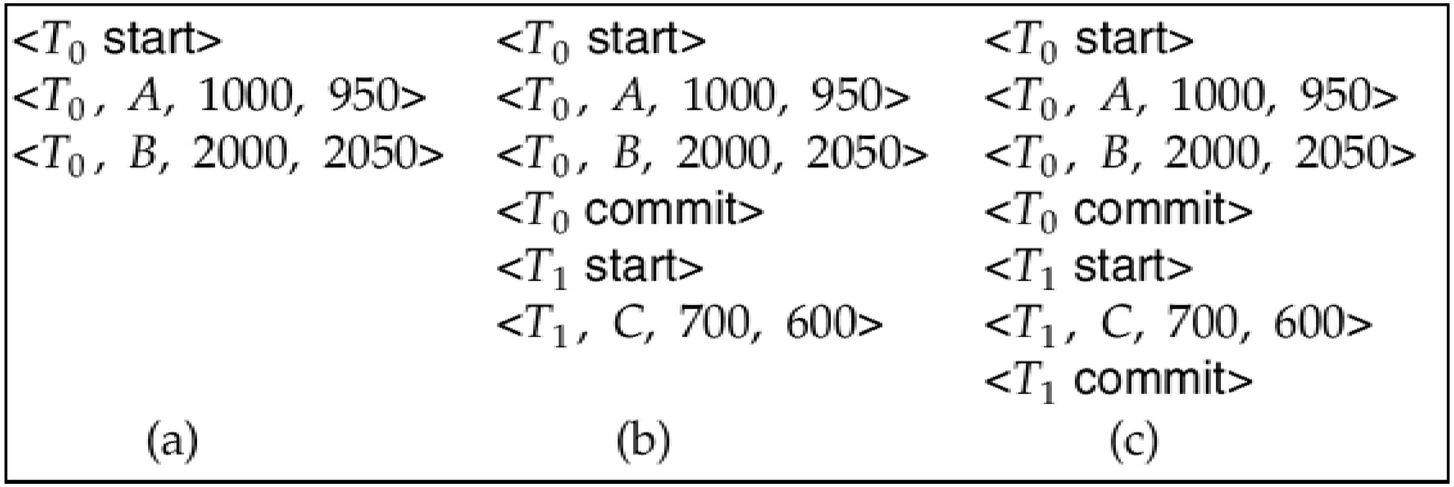
Immediate Database Modification (Cont.)

- Recovery procedure has two operations:
  - **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*
  - 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
- Recovery after a crach:
  - Undo T_i if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
  - Redo T_i if the log contains both the record <T_i **start**> and the record <T_i **commit**>.
- (Undo operations are performed before redo operations)

Immediate Database Modification (Cont.)

Example

- 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>

- If log on stable storage at time of crash is as in case:
  - (a) undo (T_0)
  - (b) undo (T_1) and redo (T_0)
  - (c) redo (T_0) and redo (T_1)

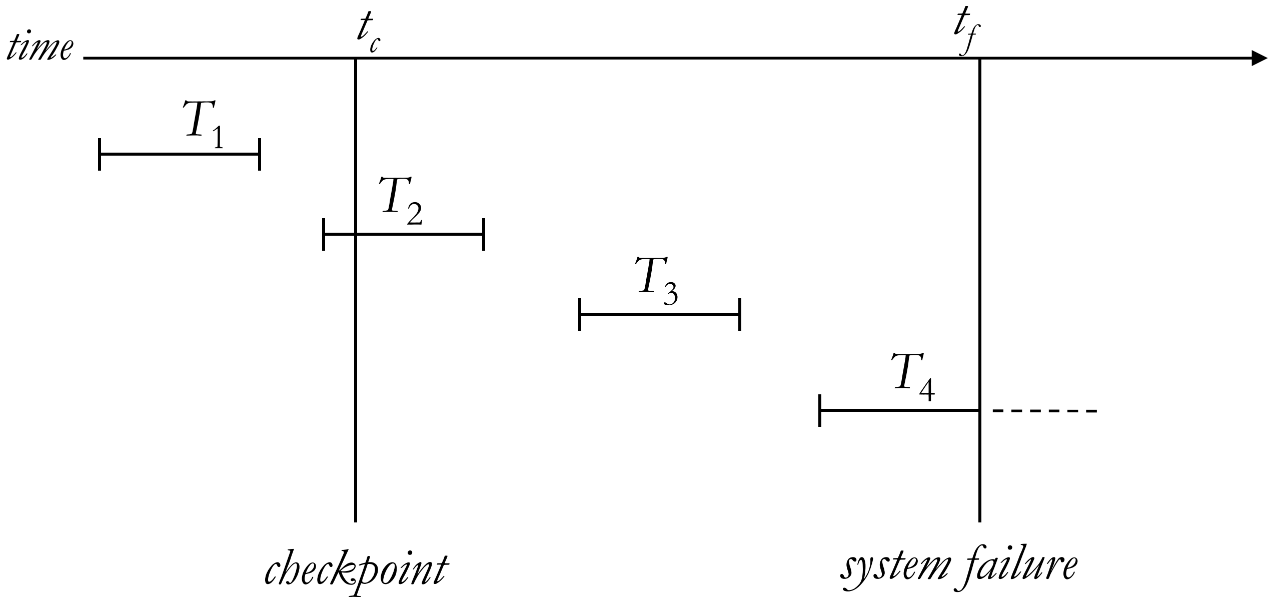
Checkpoints

- Problems in previous recovery procedures
  - searching the entire log is time-consuming
  - we might unnecessarily redo transactions which have already output their updates to the database
- Checkpoints Reduce recovery overhead
- Checkpoint process
  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**> onto stable storage

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.
- Recovery procedure
  1. Scan backwards from end of log to find the most recent <**checkpoint**>
  2. Continue scanning backwards till a record <T_i **start**> is found.
     - Need only consider the part of log following above **start** record
     - Earlier part of log can be ignored during recovery (and can be erased)
  3. For all transactions (starting from T_i or later) with no <T_i **commit**>, execute **undo**(T_i). (in case of immediate modification)
  4. Scanning forward in the log, for all transactions (starting from T_i or later) with a <T_i **commit**>, execute **redo**(T_i).

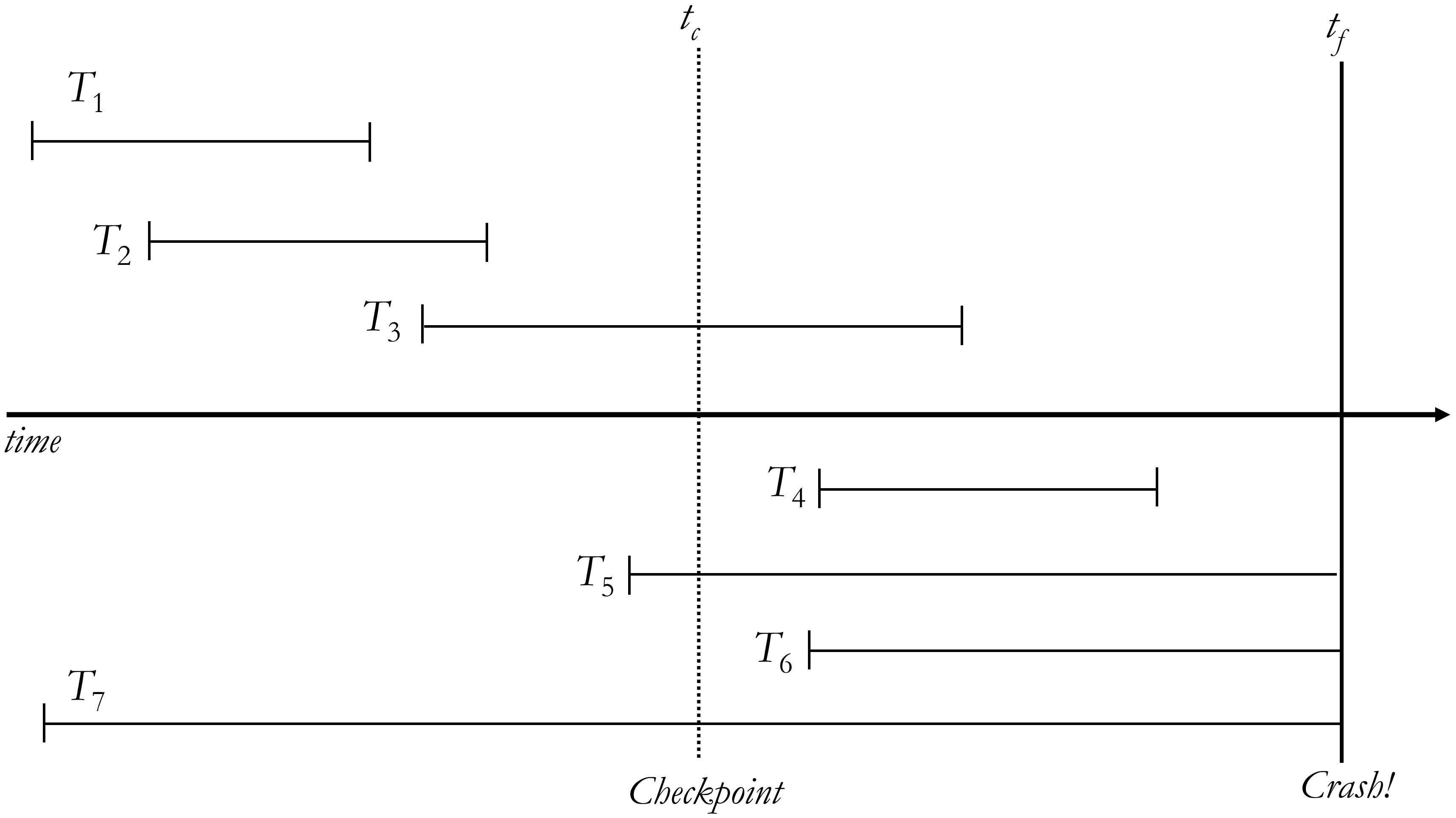
Example of Checkpoints



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

Recovery With Concurrent Transactions

- Extend the log-based recovery schemes
  - 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 concurrency control using strict two-phase locking;
  - i.e. updates of uncommitted transactions should not be visible to other transactions => recoverable
- Logging is as described earlier
  - Log records of different transactions may be interspersed in the log
- Checkpointing and recovery actions have to be changed
  - since several transactions may be active at checkpoint time


t_c
t_f
T_1
T_2
T_3
time
T_4
T_5
T_6
T_7
Checkpoint
Crash!

Recovery With Concurrent Transactions

- Checkpoints for concurrent transactions
  - Save list of active transactions at checkpoint time
  - Checkpoint record: <**checkpoint** *L*>, where *L* is the list of transactions active at the time of the checkpoint
  - The rest is identical to serial executions
- Recovery after a crash: Preparation
  1. Initialize *undo-list* and *redo-list* to empty
  2. Scan the log backwards from the end, up to the first <**checkpoint** *L*>

     For each record found during the backward scan:
     - if the record is <T_i **commit**>, add T_i to *redo-list*
     - if the record is <T_i **start**>, then if T_i is not in *redo-list*, add T_i to *undo-list*
  3. For every T_i in *L*,

     if T_i is not in *redo-list*, add T_i to *undo-list*

Recovery With Concurrent Transactions

- After the preparation phase
  - *undo-list* consists of incomplete transactions which must be undone
  - *redo-list* consists of finished transactions that must be redone
- Recovery after a crash: Recover process
  1. Scan log backwards from most recent record, until <T_i **start**> records have been found for every T_i in *undo-list.*
     - During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list.*
  2. Scan log forwards from the most recent <**checkpoint** *L*> record till the end of log
     - During the scan, perform **redo** for each log record that belongs to a transaction on *redo-list*

Example of Recovery

- Suppose the log is as shown below:

<T_0 **star**t>
<T_0, A, 0, 10>
<T_0 **commit**>
<T_1 **start**>
<T_1, B, 0, 10>
<T_2 **start**> /* Scan in Step 4 stops here */
<T_2, C, 0, 10>
<T_2, C, 10, 20>
<checkpoint {T_1, T_2}>
<T_3 **start**>
<T_3, A, 10, 20>
<T_4 **start**>
<T_4, D, 0, 10>
<T_3 **commit**>

Log Record Buffering

- Log records are buffered in main memory
  - instead of being output directly to stable storage
  - several log records can be output using a single output operation
- Log records are output to stable storage when
  - a block of log records in the buffer is full, or
  - A *log force* operation is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.

Log Record Buffering (Cont.)

Rules that must be followed

1. Log records are output to stable storage in the order in which they are created.
2. Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage
3. 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.

=> called the ***write-ahead logging*** or ***WAL*** rule

END OF CHAPTER 17