Recovery and Atomicity

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Consider transaction T_i that transfers \$50 from accoun
  A to account B; goal is either to perform all database modifications made by T_i or none at all.
- Several output operations may be required for T_i
- (to output A and B). A failure may occur after one of
- these modifications have been made but before all of
- them are made.

Recovery and Atomicity (Cont.)

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study two approaches:
  - **log-based recovery**, and
  - **shadow-paging**
- We assume (initially) that transactions run serially, that is, one after the other.

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
  <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 value of X before the write, and V_2 is the
- value to be written to X.
  - Log record notes that T_i has performed a write on data item X_j
  - X_j had value V_1 before the write, and will have value V_2 after the write.
- When T_i finishes it last statement, the log record <T_i **commit**> is written.
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Deferred Database Modification

- The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing $<T_i$ ***start***$>$ record to log.
- A **write**(X) operation results in a log record $<T_i, X, V>$ being written, where V is the new value for X
  - Note: old value is not needed for this scheme
- The write is not performed on X at this time, but is deferred.
- When T_i partially commits, $<T_i$ **commit**$>$ is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.

Deferred Database Modification (Cont.)

- During recovery after a crash, a transaction needs to be redone if
- and only if both <T_i **start**> and<T_i **commit**> are there in the log.
- Redoing a transaction T_i (**redo**T_i) sets the value of all data items
- updated by the transaction to the new values.
- Crashes can occur while
  - the transaction is executing the original updates, or
  - while recovery action is being taken
- 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.)

- 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, 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>

- 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) must be performed since <T_0 **commit**> is present
  (c) **redo**(T_0) must be performed followed by redo(T_1) since <T_0 **commit**> and <T_i commit> are present

Immediate Database Modification

- The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output**(B) operation for a data block B, all log records corresponding to items B must be flushed to stable storage
- 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

Immediate Database Modification Example

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

- Note: B_X denotes block containing X.

Immediate Database Modification (Cont.)

- Recovery procedure has two operations instead of one:
  - **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**
  - That is, even if the operation is executed multiple times the effe is the same as if it is executed once
    - Needed since operations may get re-executed during recove
- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
  - Transaction T_i needs to be redone if the log contains both the
  - record <T_i **start**> and the record <T_i **commit**>.
- Undo operations are performed first, then redo operations.

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

<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>
(a)	(b)	(c)

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

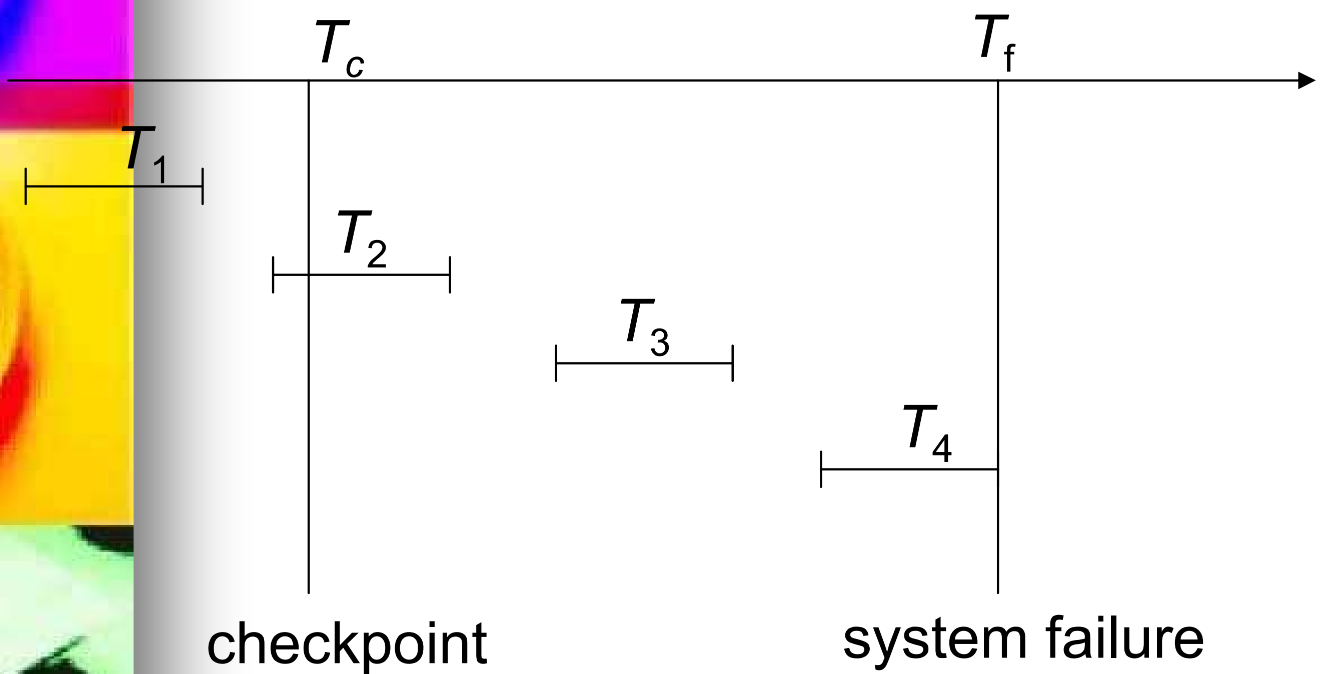
(b) undo (T_1) and redo (T_0): C is restored to 700, and then A and B ar
set to 950 and 2050 respectively.

(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

- Problems in recovery procedure as discussed earlier :
  1. searching the entire log is time-consuming
  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**> onto stable storage.

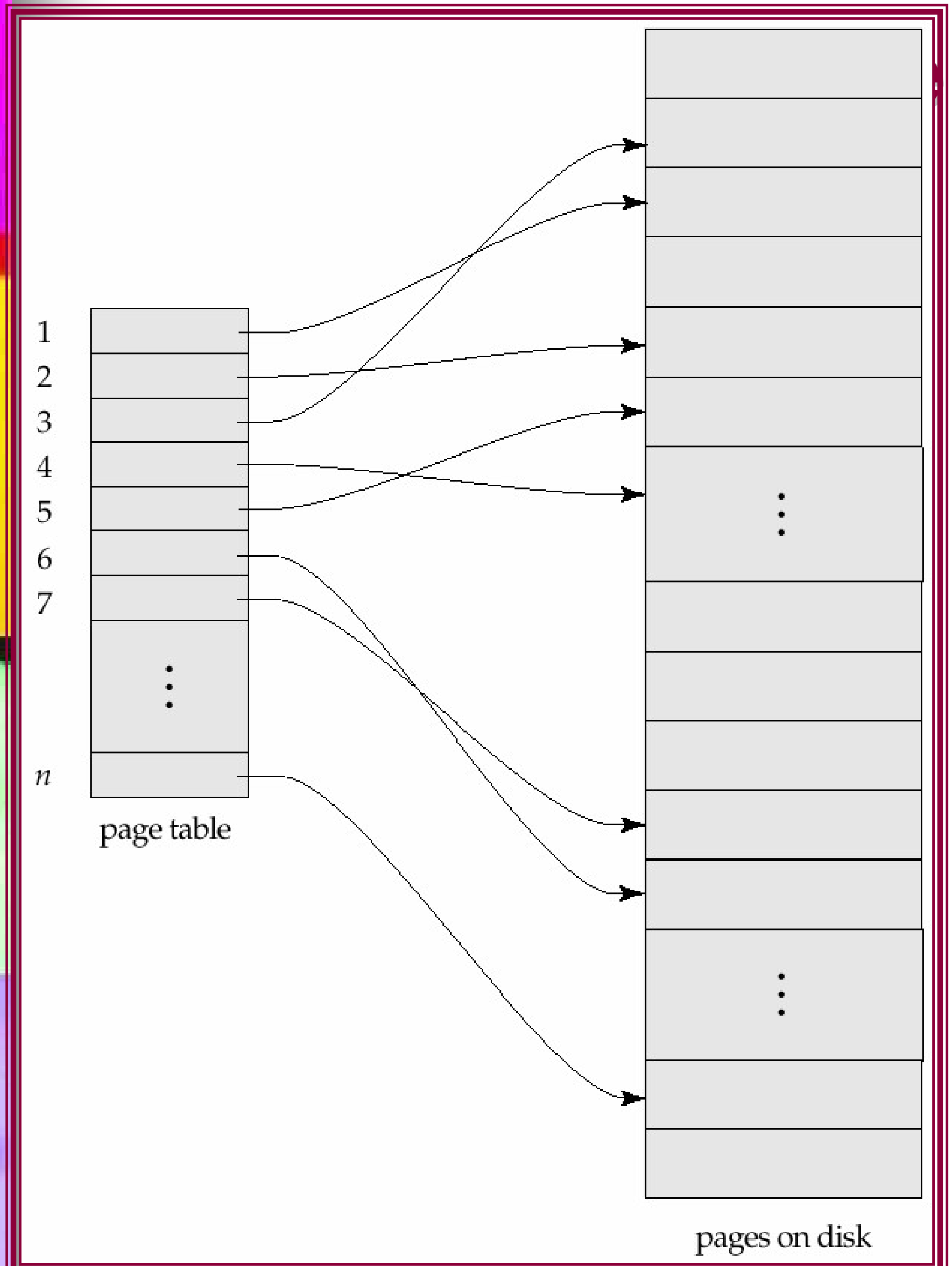
Example of Checkpoints



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

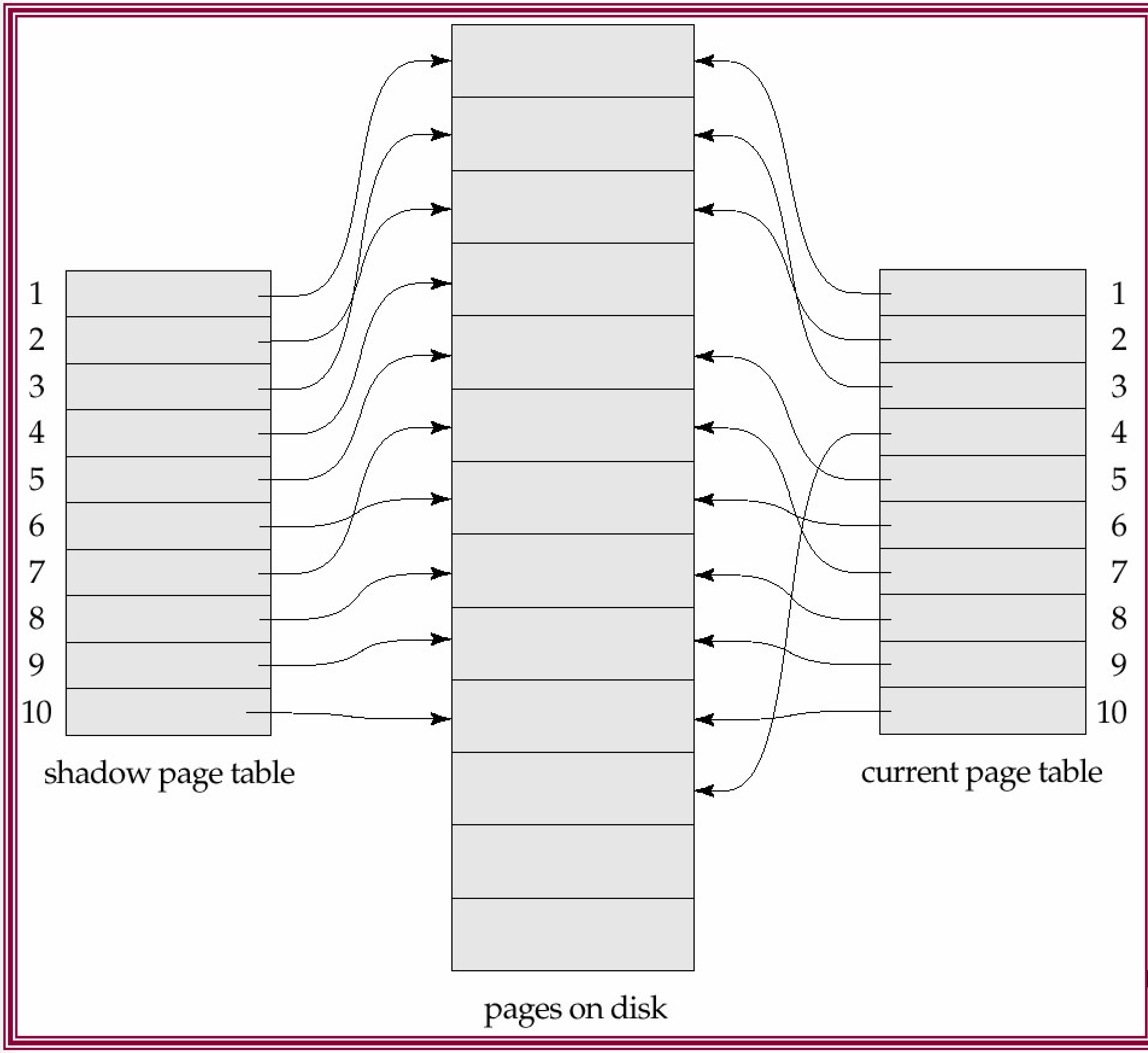
Shadow Paging

- **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 transactio
- –the **current page table**, and the **shadow page table**
- Store the shadow page table in nonvolatile storage, such that stat
- 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 pag
- table is used for data item accesses during execution of the transa
- 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


1
2
3
4
5
6
7
n
page table
pages on disk

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).

Show 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
    - Can be reduced by using a page table structured like a B^+-tree
      - No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
  - 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 contai old versions of modified data need to be garbage collected
  - Hard to extend algorithm to allow transactions to run concurre
    - Easier to extend log based schemes

THE END