Recovery and Atomicity

- Consider transaction T_i that transfers \$50 from account 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.
- 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.
- 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)

Deferred Database Modification

- This scheme ensures atomicity despite failures by recording all modifications to log, but deferring 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. 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, log records are 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 $<T_1$ **commit**$>$ are present

Immediate Database Modification

- This 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
- 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 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

- 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
- 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.

(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.

(b) **undo**(T_1) and **redo**(T_0): C is restored to 700, and then A and B are 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.

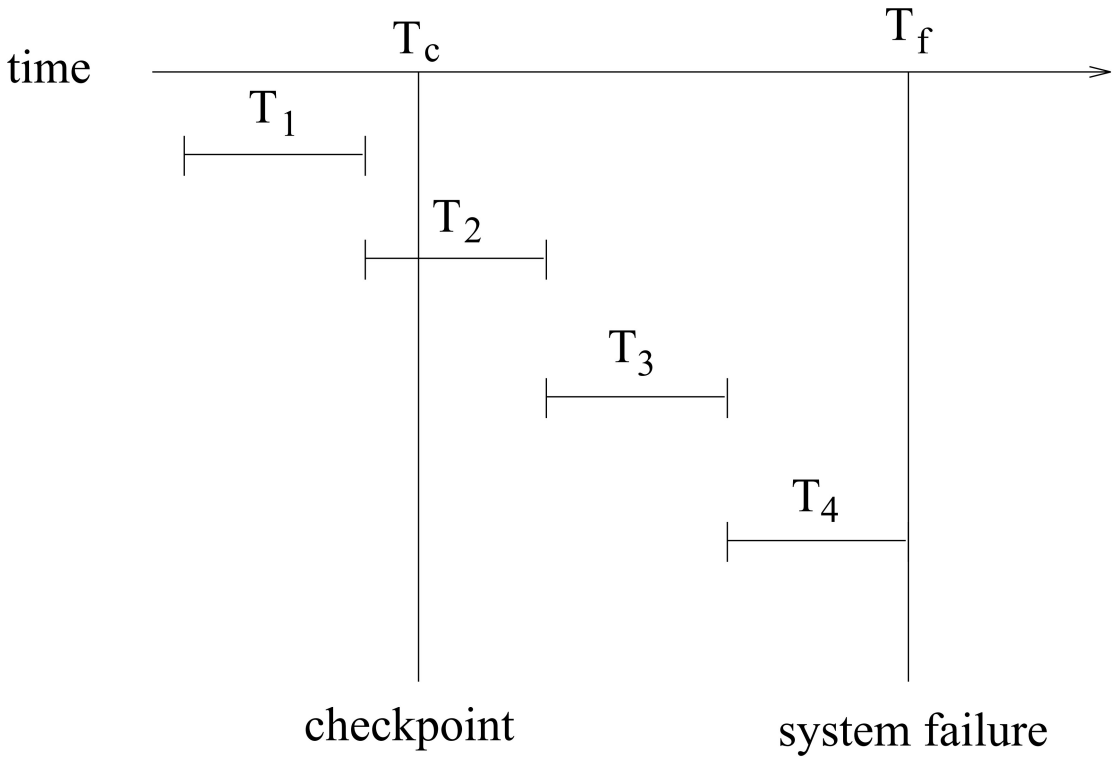
Checkpoints

- 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.

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.
- Scan backwards from end of log to find the most recent $<$**checkpoint**$>$ record
- 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 whenever desired.
- For all transactions (starting from T_i or later) with no $<T_i$ **commit**$>$, execute **undo**(T_i). (Done only in case of immediate modification.)
- 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

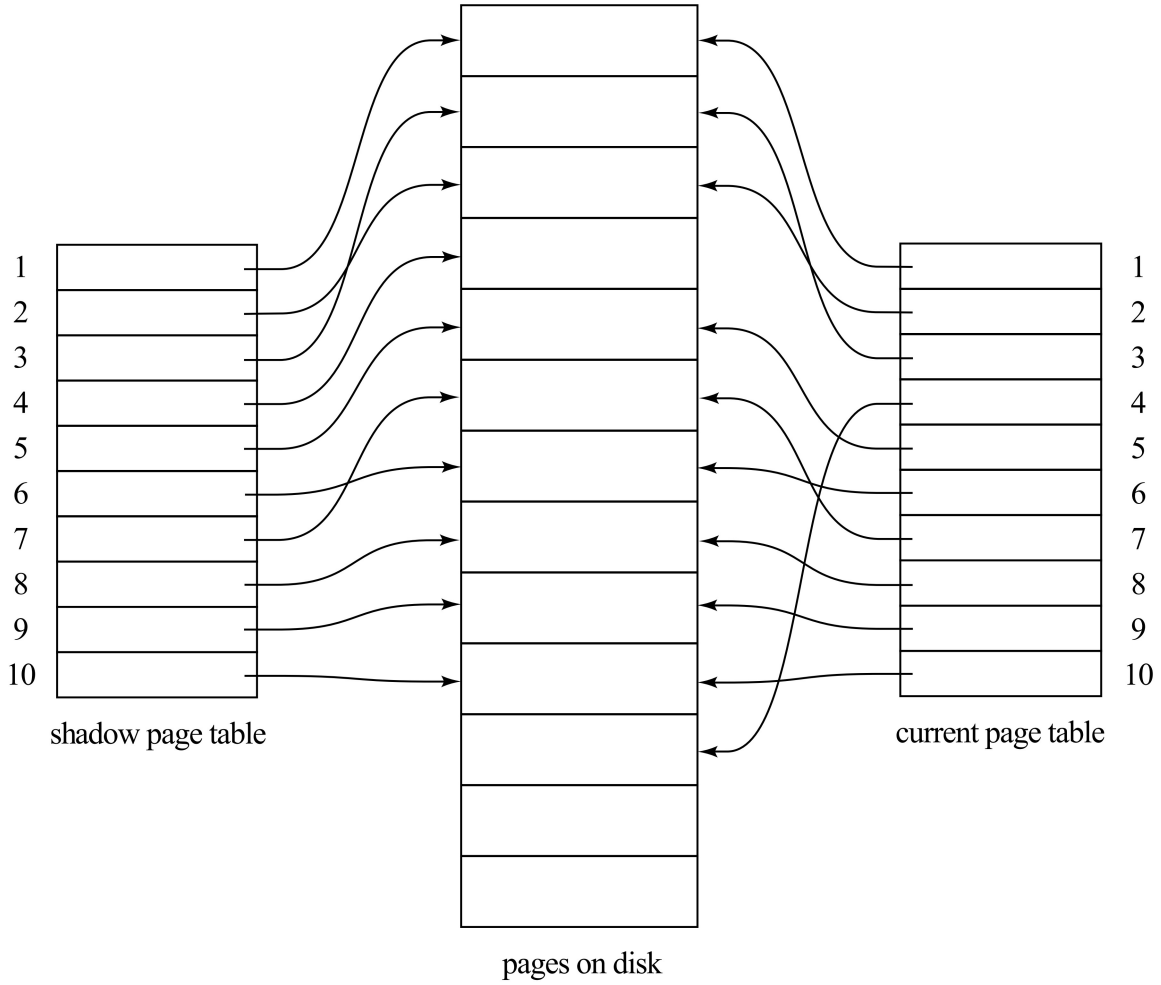


- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone
- T_4 undone

Shadow Paging

- Alternative to log-based recovery
- Idea: maintain *two* page tables during the lifetime of a transaction – the *current* page table, and the *shadow* page table
- Store the shadow page table in nonvolatile storage, such that state 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 page table is used for data item accesses during execution of the transaction.
- 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, and the update is performed on the copy

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 the new shadow page table
     - 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).

Shadow Paging (Cont.)

- Advantages of shadow-paging over log-based schemes – no overhead of writing log records; recovery is trivial
- Disadvantages :
  - Commit overhead is high (many pages need to be flushed)
  - Data gets fragmented (related pages get separated)
  - After every transaction completion, the database pages containing old versions of modified data need to be garbage collected and put into the list of unused pages
  - Hard to extend algorithm to allow transactions to run concurrently

Recovery With Concurrent Transactions

- We modify the log-based recovery schemes to allow multiple transactions to execute concurrently.
  - 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; ie. the updates of uncommitted transactions should not be visible to other transactions
- Logging is done as described earlier. Log records of different transactions may be interspersed in the log.
- The checkpointing technique and actions taken on recovery have to be changed, since several transactions may be active when a checkpoint is performed.

Recovery With Concurrent Transactions (Cont.)

- Checkpoints are performed as before, except that the checkpoint log record is now of the form $<$**checkpoint** $L>$, where L is the list of transactions active at the time of the checkpoint.
- When the system recovers from a crash, it first does the following:
  1. Initialize *undo-list* and *redo-list* to empty
  2. Scan the log backwards from the end, stopping when the first $<$**checkpoint** $L>$ record is found. For each record found during the 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 (Cont.)

- At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.
- Recovery now continues as follows:
  4. Scan log backwards from most recent record, stopping when $< T_i \ \mathbf{start} >$ records have been encountered for every T_i in *undo-list*.
     During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*.
  5. Locate the most recent <**checkpoint** *L*> record.
  6. Scan log forwards from the <**checkpoint** *L*> record till the end of the log.
     During the scan, perform **redo** for each log record that belongs to a transaction on *redo-list*.

Example of Recovery

Go over the steps of the recovery algorithm on the following log:

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