CS 5/7330

Recovery Overview

Recovery – why?

- **A**CI**D** properties
  - Atomicity: all-or-nothing
  - Durability: Once committed, changes must be permanent
- If the system is always functional, that's ok.
- However, system may crash (failure).

Recovery – why?

- Types of failures:
  - **Transaction failure** :
    - **Logical errors**: transaction cannot complete due to some internal error condition (e.g. your program have a division by zero error)
    - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)

Recovery – why?

- Types of failures (ctd)
  - **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 be corrupted by system crash
      - Database systems have numerous integrity checks to prevent corruption of disk data

Recovery – why?

- Types of failures (ctd):
  - **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
    - With some disk systems, some failures maybe correctable.
- We focus on transaction/system failure as well as other failures that are correctable.

Recovery – why?

- Failure can occur ANY time
  - Including the time that you do not want
  - We do assume in this class that writing a page is atomic (i.e. failure do not occur in the middle of writing a page)
- Thus need to maintain atomicity and durability at all times

Recovery – why?

- After failure, the system needs to recover
  - Need to recover to a consistent state
  - Need to ensure all transactions are atomic and durable
- Thus when the system go back up and running after a failure
  - A recovery module is up and running – before allowing any other transaction to run
  - It will restore the database to the consistent state as well as ensure the ACID properties
  - After that transactions may continue
- Notice that information need to be stored during the normal running of the transaction for the recovery module to run properly.

Recovery – atomicity requirement

- Suppose the system crash while a transaction T is being executed (but not committed)
- During recovery, one need to
  - Find those transactions
  - Ensure atomicity is held
    - Abort or complete?
    - Abort! (in many case, don't know how to complete anyway)
    - Abort means restoring the database to the point where those transactions has not started (as some changes may have propagated to the disk)
    - Thus, after the recovery procedure, those transaction should seems to have NEVER been executed

Recovery – atomicity requirement

- Counter-argument:
  - Can we just not write anything (i.e. uncommitted data) to the disk until T commits?
- Problems:
  - When pages are to be read/written, it is moved from the disk to a buffer
  - If we avoid writing anything to the disk immediately, then the buffer holding that page cannot be released until end of T
  - If T is long, or if T write something early in the transaction, it can hold up resources
  - If many transactions are running, buffers can be used up very quickly
- If the uncommitted data is written to the disk, then buffer replacement policy (e.g. FIFO, LRU) can utilize resources much more effectively and allow higher concurrency

Recovery – atomicity requirement

- Thus, during recovery, one need to
  - Find those transactions that has started but not yet committed
  - Ensure atomicity is held
    - Find all the changes to the database that the transactions has done
    - *Undo* all the changes

Recovery – durability requirement

- Suppose the crash occurs right after a transaction T committed.
- Seems to be ok, but …
  - T may have written something onto the disk
  - However, the writes may not have propagated to the disk
  - Reasons
    - The writes may be scheduled/buffered but the system crashed before such writes can execute
    - The buffer manager/virtual memory management may decide to put a disk page into main memory for a long time (save disk access time)
      - This is especially true on a network/shared file system

Recovery – durability requirement

- Remember, when you issue a write() command, the system does not necessary write what you have onto disk immediately.
- In some systems, you may issue a flush() command to force the writes onto the disk

Recovery – durability requirement

- Counter-argument
  - Don’t commit until transaction is certain that all the pages are safely written onto the disk
- Problems:
  - Sometimes it is impossible to check for that
  - Even if it is possible, it may slow down the transaction significantly
  - As well as lead to lower concurrency (locks being held longer)

Recovery – durability requirement

- Thus, during recovery, one need to
  - Find those transactions that has committed
  - Ensure durability is held
    - Check if all the changes made by the transactions is written onto the disk
    - If not, then *redo* all the changes

Force/No-force vs. Steal/No-steal

- To sum up, what we need to do at recovery depends on how we deal with uncommitted data (especially data in buffers)
- **Steal/No-steal**: A system is said to steal buffers if it allows buffers that contain dirty (uncommitted but updated) data to be swapped to physical storage
  - If steal is allowed, undo is necessary
- **Force/No-force**: A system is said to force buffers if every committed data is guaranteed to be forced onto the disk at commit time
  - If force is not allowed, redo is necessary

Force/No-force vs. Steal/No-steal

- Thus no-steal and force means no work required at recovery.
- However, it is either unimplementable or impractical
- Thus we need to deal with them

Recovery -- overview

- 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
- In order to achieve (1), we need to store the information in stable storage

Stable storage

- **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 mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media

Stable storage

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

Stable storage

- Copies of a block may differ due to failure during output operation. To recover from failure:
  1. First find inconsistent blocks:
     1. *Expensive solution*: Compare the two copies of every disk block.
     2. *Better solution*:
        - Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
        - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
        - Used in hardware RAID systems
  2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

Log files

- Logs are needed to record the operations on the database during normal operations
- The log is a sequence of **log records**, and maintains a record of update activities on the database.

Log files

- Different types of log records during normal operations:
  - Begin record: <T_i, start> -- registered when transaction T_i begins
  - Write record: <T_i, X, V_{old}, V_{new}> -- registered when a database item X is updated by T_i, where V_{old}, V_{new} store the old and new values respectively
    - V_{old}, V_{new} also known as before-image & after-image respectively
  - Commit record: <T_i, commit> -- registered when T_i commits.
    - Formally, a transaction commits when the commit log record is written
  - Abort record: <T_i, abort> -- registered when T_i aborts

Log files

- With such information, redoing and undoing operations can be done:
  - Undo: copy V_{old} back to the object.
  - Redo: copy V_{new} back to the object

Log files

- Logs are written onto stable storage
- Question: given an operation to be logged, should we log first or execute first?
- Consider the case the system fail between the two operations

Log files

- Suppose we execute first, and before we can log
  - Then when the system recovers, it cannot find the write operation on the log.
  - If the operation is from a transaction that is not committed, then one do not know that the operation needs to be undone
  - Big problem!

Log files

- Suppose we log first and system crash before it executes
  - Then when the system recovers, it saw an operation that have not been executed
  - If the operation is from a transaction that is not committed, then one is undoing an operation that has not been done.
  - Undo means copying old value back to the database
  - In this case, overwriting the same value (since new value never make it to database)
  - No problem. (Maybe inefficient, but correct)

Log files

- Thus logs must be **write-ahead logs** (WAL)
  - i.e. all operations must be logged first
  - Log records must be forced to stable storage before actually operations can be executed

Log-based recovery : basic approach

- Given a write-ahead log. How should recovery proceed after the system crash?
- Two major steps:
  - Locating the transaction that need works to be done
  - Apply *compensatory* action on these transactions

Log-based recovery : basic approach

- Step 1: locating transactions that needed to be dealt with.
  - Uncommitted but active transactions
    - Need undo
    - Transactions that has <start T> in log, but not <commit T>
  - Committed transactions
    - Need redo
    - Transactions that has <commit T> in log

Log-based recovery : basic approach

(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>

- Example
  a) Undo T_0
  b) Undo T_1, Redo T_0
  c) Redo T_0, T_1

Log-based recovery : basic approach

- Step 2: Apply compensatory actions
  - Redo & Undo
  - Requirement: actions have to be **idempotent**
    - That is, even if the operation is executed multiple times the effect is the same as if it is executed once

Log-based recovery : basic approach

- Why idempotent?
- Consider the case when system crash during recovery
- Assume we do not log the compensatory actions
- Then after the system go back up again we will apply the same operations.
- Thus the need to be idempotent

Log-based recovery : basic approach

- For undo: copying old value of the item from the log to the database
- For redo: copying new value of the item from the log to the database
- Both operations are idempotent

Log-based recovery : basic approach

- Order of operations: redo first or undo first?
- Depends on the recovery method used.
- Undo first then redo is typically correct.
- Undo has to go backwards (from end of the log to beginning)
- Redo has to go forward (from beginning of log to the end)
- Why?

Checkpoints

- If the system have been running for a while, the log can be huge.
- Some committed transactions may have result forced to the disk already
- Thus need to create checkpoints to eliminate redo a large number of transactions

Checkpoints

- The simplest checkpoint is a hard checkpoint
  - Normal operation of database is halted
  - All data that is in the buffers are forced (flushed) to the disk
  - A <checkpoint> record is written on the log (write-ahead)
  - Normal operation of database resumes

Recovery with checkpoints

- Consider a log with a checkpoint
- For recovery purpose:
  - Finding transactions that need to be redone:
    - i.e. if <commit T> record is in the log
    - Now, if <commit T> appears before <checkpoint> record.
      - We do not need to redo it, why?
  - Finding transactions that need to be undone:
    - i.e. if <start T> record is in the log, but <commit T> isn't
    - Now, if <start T> is in the log before the <checkpoint> record but <commit T> is not in the log...
      - We still need to undo it, why?

Recovery with checkpoints

- Thus, recovery procedure after a system crash
  1. Start reading the log from the end
  2. Before reaching a <checkpoint> record
     - If one see a <commit T> record, put it in the list of transactions to redo (redo-list)
     - If one see a <start T> record, such that T is not already in the redo-list, then put it in the list of transactions to undo (undo-list)
  3. After reaching a <checkpoint> record, continue to scan backwards
     - If one see a <commit T> record, put it in the list of transaction that is done (done-list)
     - If one see a <start T> record, such that T is not already in the redo-list or the done-list, then put it in the undo-list

Recovery with checkpoints

4. Once the whole log is read, the system know which transaction to undo and redo.
5. Start reading the log backwards again
   - If a <write> record belong to a transaction in the undo-list, undo this operation (by overwriting the item with the old value)
   - Until the beginning of the log is reached
6. Start reading from the <checkpoint> record.
   - If a <write> record belong to a transaction in the redo-list, redo this operation (by overwriting the item with the new value)

Recovery with checkpoints

- One way of speeding up
  - One still have to go through all the logs to find all transactions that need to be undone
  - To speed up, when checkpointing, one can write the list of all transactions that are active with the checkpoint
  - i.e. <checkpoint T1, T2, … Tn> where Ti are all the transactions that are running at the time of checkpointing
  - Save time in the first stage (but not necessarily in the second)
- Other optimizations to be discussed later

Recovery with checkpoints -- example

1. <start T1>
2. <T1 write X, 30, 40>
3. <start T2>
4. <T1 write Y, 40, 50>
5. <start T3>
6. <T3 write Z, 8, 30>
7. <commit T3>
8. <start T4>
9. <T4 write A 10, 5>
10. <checkpoint T1, T2, T4>
11. <T1 write Z 30, 10>
12. <T2 write W 50, 9>
13. <commit T1>

System crashes!

- Consider the log on the LHS
- First, determine which transactions need to be redo/undo
- Initially
  - Undo = {}
  - Redo = {}
  - Done = {}

Recovery with checkpoints -- example

1. <start T1>
2. <T1 write X, 30, 40>
3. <start T2>
4. <T1 write Y, 40, 50>
5. <start T3>
6. <T3 write Z, 8, 30>
7. <commit T3>
8. <start T4>
9. <T4 write A 10, 5>
10. <checkpoint T1, T2, T4>
11. <T1 write Z 30, 10>
12. <T2 write W 50, 9>
13. <commit T1>

System crashes!

- T1 commits after checkpoint
- Add T1 to redo
  - Undo = {}
  - Redo = {T1}
  - Done = {}

Recovery with checkpoints -- example

1. <start T1>
2. <T1 write X, 30, 40>
3. <start T2>
4. <T1 write Y, 40, 50>
5. <start T3>
6. <T3 write Z, 8, 30>
7. <commit T3>
8. <start T4>
9. <T4 write A 10, 5>
10. <checkpoint T1, T2, T4>
11. <T1 write Z 30, 10>
12. <T2 write W 50, 9>
13. <commit T1>

System crashes!

- T2 active but not committed
- Add T2 to undo
  - Undo = {T2}
  - Redo = {T1}
  - Done = {}

Recovery with checkpoints -- example

1. <start T1>
2. <T1 write X, 30, 40>
3. <start T2>
4. <T1 write Y, 40, 50>
5. <start T3>
6. <T3 write Z, 8, 30>
7. <commit T3>
8. <start T4>
9. <T4 write A 10, 5>
10. <checkpoint T1, T2, T4>
11. <T1 write Z 30, 10>
12. <T2 write W 50, 9>
13. <commit T1>

System crashes!

- T1 already in Redo list
  - Undo = {T2}
  - Redo = {T1}
  - Done = {}

Recovery with checkpoints -- example

1. <start T1>
2. <T1 write X, 30, 40>
3. <start T2>
4. <T1 write Y, 40, 50>
5. <start T3>
6. <T3 write Z, 8, 30>
7. <commit T3>
8. <start T4>
9. <T4 write A 10, 5>
10. <checkpoint T1, T2, T4>
11. <T1 write Z 30, 10>
12. <T2 write W 50, 9>
13. <commit T1>

System crashes!

- Checkpoint reached
- T4 added to undo
  - Undo = {T2, T4}
  - Redo = {T1}
  - Done = {}
- All the necessary transactions discovered

Recovery with checkpoints -- example

1. <start T1>
2. <T1 write X, 30, 40>
3. <start T2>
4. <T1 write Y, 40, 50>
5. <start T3>
6. <T3 write Z, 8, 30>
7. <commit T3>
8. <start T4>
9. <T4 write A 10, 5>
10. <checkpoint T1, T2, T4>
11. <T1 write Z 30, 10>
12. <T2 write W 50, 9>
13. <commit T1>

System crashes!

- Notice that T3 does not need to be redo because T3 is committed before checkpoint
- Thus all the pages T3 changes is forced onto the disk

Recovery with checkpoints -- example

1. <start T1>
2. <T1 write X, 30, 40>
3. <start T2>
4. <T1 write Y, 40, 50>
5. <start T3>
6. <T3 write Z, 8, 30>
7. <commit T3>
8. <start T4>
9. <T4 write A 10, 5>
10. <checkpoint T1, T2, T4>
11. <T1 write Z 30, 10>
12. <T2 write W 50, 9>
13. <commit T1>

System crashes!

- Step 2 : Undo
- T2, T4 need to be undone
- Undo are done backwards
- Read log backwards until the <start T> for all transactions to be undone reached

Recovery with checkpoints -- example

1. <start T1>
2. <T1 write X, 30, 40>
3. <start T2>
4. <T1 write Y, 40, 50>
5. <start T3>
6. <T3 write Z, 8, 30>
7. <commit T3>
8. <start T4>
9. <T4 write A 10, 5>
10. <checkpoint T1, T2, T4>
11. <T1 write Z 30, 10>
12. <T2 write W 50, 9>
13. <commit T1>

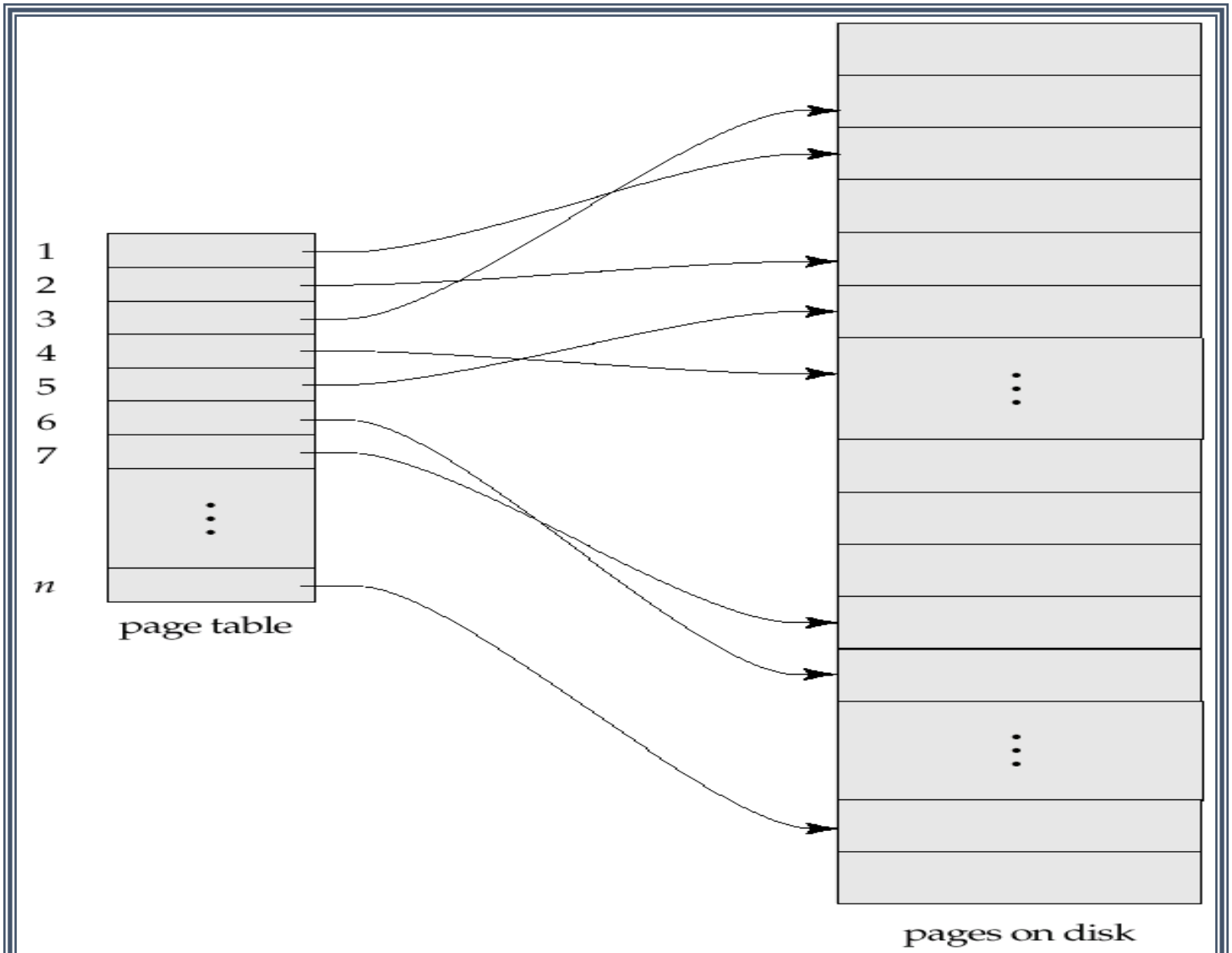
System crashes!

- Step 3 : Redo
- T1 needed to be redo
- So logs have to be returned from the earliest start time of all transactions to be redone

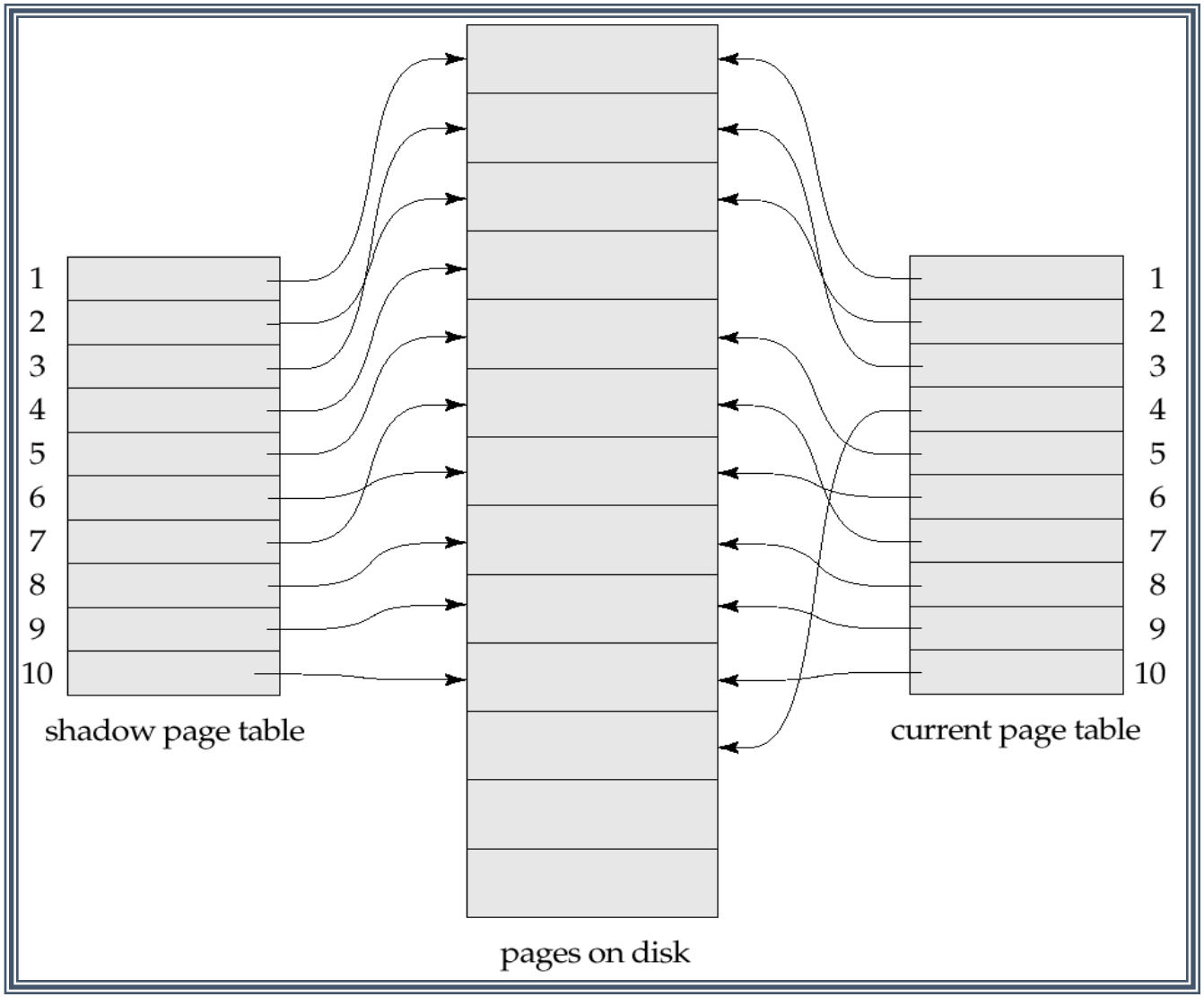
Non log-based recovery: shadow paging

- **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 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
  - The update is performed on the copy

Non log-based recovery: shadow paging



Non log-based recovery: shadow paging



Shadow paging

- 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).

Shadow paging

- 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 containing old versions of modified data need to be garbage collected
  - Hard to extend algorithm to allow transactions to run concurrently
    - Easier to extend log based schemes