Chapter 16: Recovery System

Database System Concepts, 6th Ed.

Chapter 16: Recovery System

- In case of any failure, information may be lost, so DBMS must take actions in advance to ensure that the Atomicity and Durability properties of transactions

Goal

To restore the database to the consistent state that existed before the failure, with high availability (minimize the time to restore)

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Remote Backup Systems

Failure Classification

- Transaction failure
- System crash
- Disk failure

Failure Classification

- **Transaction Failure**
  - **Logical Errors**
    - Transaction cannot complete due to some internal error condition such as bad input, data not found, overflow, or resource limit exceeded
  - **System Errors**
    - The database system must terminate an active transaction due to an error condition (e.g., deadlock) as transaction cannot continue with its normal execution, but the transaction can be re-executed at later time

Failure Classification

- **System Crash**
  - Due to the power failure or other hardware malfunction or bug in the database software or operating system
  - **Fail-stop assumption**
    - Assumption that this system crash causes the loss of content of volatile storage and brings transaction processing to a halt, but do not corrupt non-volatile storage contents
    - Database systems have numerous integrity checks to prevent corruption of disk data

Failure Classification

- **Disk Failure**
  - A head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable, as disk drives use checksums to detect failures

Recovery Algorithms

- Consider transaction T_i that transfers \$50 from account *A* to account *B*
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database
  - **A failure may occur after one of these modifications** have been made but before both of them are made
  - Modifying the database **without ensuring that the transaction will commit** may leave the database in an inconsistent state
  - Not modifying the database may result in lost updates if failure occurs **just after transaction commits**

Recovery Algorithms

- 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, But may still fail, losing data
  - 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
    - Replicate the needed information with independent failure modes and to update the information in a controlled manner without damage of needed information

Stable-Storage Implementation

- 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
  - As when **block transfers between main memory and disk can result in**
    - Successful completion
    - Partial failure: at the midst of transfer, destination block has incorrect information
    - Total failure: destination block was never updated

Stable-Storage Implementation

- If data-transfer failure occurs,
  - System should detect it and invoke a recovery procedure to restore the block to a consistent state
    - To do so, maintains two physical blocks for each logical database blocks
      - If Mirrored disks, blocks are at the same location
      - If Remote backup, one block is local, another is at remote site
    - Execute output operation as,

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 Implementation (Cont.)

- **Copies of a block may differ due to failure during output operation, to recover from failure**:
  - System examines two copies of blocks, if
    - Both are same and no detectable error exist, No further actions
      - Detectable errors in a disk block, such as a partial write to the block, detected by storing a checksum with each block
    - Error in one block
      - Then replaces its content with the content of the other block
    - No detectable error but differ in content
      - Then replaces the content of the first block with the value of the second and recovery procedure ensures that a write to stable storage either succeeds completes (updates all copies) or results in no change

Stable-Storage Implementation (Cont.)

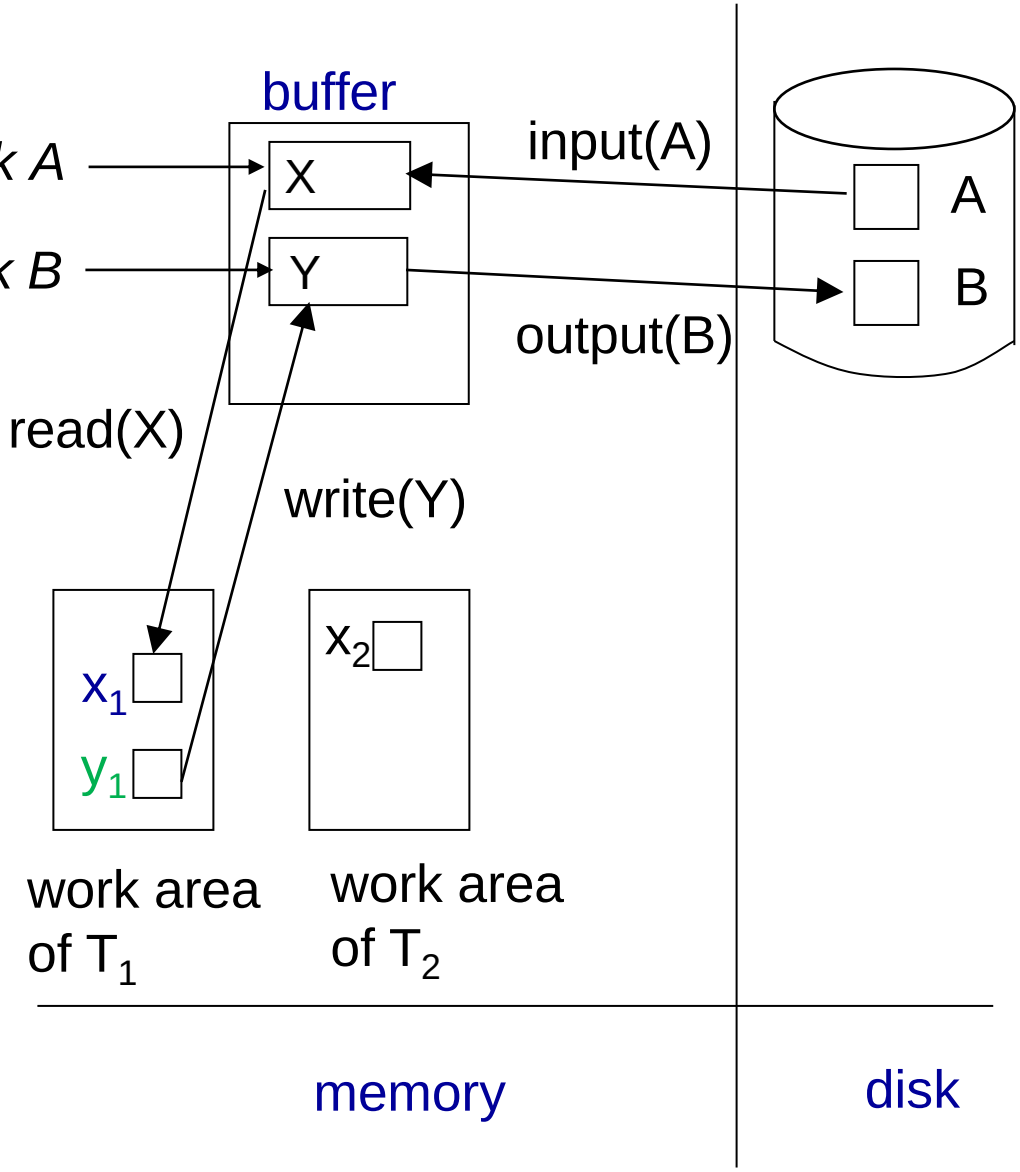
- Comparison of two copies of blocks is expensive, to reduce cost
  - Keep track of block writes that are in progress
    - Utilizes small amount of nonvolatile RAM or special area of disk
    - This information is used during recovery to find blocks for which writes were in progress (may be inconsistent) and only compare copies of these
      - Used in hardware RAID systems

Data Access

- **Blocks**
  - Fixed length storage unit of database for data transfer to and from disk
    - Contains several data items
    - Here, assume for simplicity, that each data item fits in, and is stored inside, a single block
  - Transaction input block information from disk to main memory and then output the block information back onto disk
  - **Physical blocks -** blocks residing on the disk
  - **Buffer blocks** - blocks residing temporarily in main memory
- **Disk buffer** - area of memory where blocks reside temporarily
- **Work area -** Each transaction has a private work area in which copies of data items accessed and updated are kept created on transaction initiation and removed when commit/abort by system

Example of Data Access

- Block movements initiated through two operations:
  - **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



Data Access (Cont.)

- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept
  - T_i's local copy of a data item X is called x_i
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read**(X) assigns the value of data item X to the local variable x_i
  - **write**(X) assigns the value of local variable x_i to data item {X} in the buffer block
  - **Note: output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit
- Transactions
  - Must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
  - **write**(X) can be executed at any time before the transaction commits

Recovery and Atomicity

- To ensure atomicity despite failures, first output information describing the modifications to stable storage without modifying the database itself
- Here, studying **log-based recovery mechanisms** in detail
  - First present key concepts
  - And then present the actual recovery algorithm
- Less used alternative: **shadow-paging** (brief details in book)

Log-Based Recovery

- **Log**
  - Widely used structure for recording database modifications
  - A sequence of **log records**, and maintains a record of update activities on the database
- **Update Log Record Fields**
  - Transaction Identifier for transaction with write operation
  - Data-item Identifier of the data item written
    - Location on disk of the data item, consisting of the block identifier of the block on which the data item resides and an offset within the block
  - Old Value - the value of the data item prior to the write
  - New Value - the value of the data item after write

Log-Based Recovery

Types of log records

1. When transaction T_i starts, it registers itself by writing a log record
   $<T_i$ **start**$>$
2. *Before* T_i executes **write**(X), a log record
   $<T_i$, ***X, V_1, V_2***$>$
   V_1 is the value of X before the write (**old value**)
   V_2 is the value to be written to X (the **new value**)
3. When T_i finishes it last statement, the log record
   $<T_i$ **commi**t$>$
4. If T_i is aborted, the log record
   $<T_i$ **abort**$>$

Log reside in stable storage

- Every log record is written to the end of the log on stable storage as soon as it is created

Log-Based Recovery

- Database Modification
  - Performing update on a disk buffer or on the disk itself
  - Transaction creates a log record prior to modifying the database
    - Whenever a transaction performs a write, the log record for that write be created and added to the log, before the database is modified
    - Once a log record exists, output the modification to the database if that is desirable
    - Also, provides ability to undo the modification that has already been output to the database using old value of log record

Log-Based Recovery

- Database Modification
  - **Two approaches for database modification using logs**
    1. Deferred database modification
       - Updates to buffer/disk only at the time of transaction commit
       - Simplifies some aspects of recovery
       - But
         - Overhead of storing local copy of all updated data items
         - If a transaction reads a data item that it has updated, it must read the value from its local copy
    2. Immediate database modification

Log-Based Recovery

- Database Modification
  - **Two approaches for database modification using logs**
    1. Deferred database modification
       - Only at the time of transaction commit
    2. Immediate database modification
       - Updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits/still active
       - Optimized to reduce overhead

Database Modification

- **Database Modification Operations using log**
  - **Undo** sets the data item specified in the log record to the old value
  - **Redo** sets the data item specified in the log record to the new value

Transaction Commit

- Here considers **Immediate Database Modification**
- A transaction is said to have committed when its commit log record is output to stable storage
  - With all previous log records of the transaction must have been output already
  - Thus, enough information in log is available to ensure the recovery from the system crash
- Example
  - T_0 transfers \$50 from Account A to Account B
  - T_1 withdraws \$100 from Account C

Immediate Database Modification Example

Log	Database
<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
<T_1 **commit**>	

Example

- T_0 transfers $50 from Account A to Account B
- T_1 withdraws $100 from Account C

Undo and Redo Operations

- **Undo** of a log record <T_i, X, V_1, V_2> writes the **old** value V_1 to X
- **Redo** of a log record <T_i, X, V_1, V_2> writes the **new** value V_2 to X
- **Undo and Redo of Transactions**
  - **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**
    - Each time a data item X is restored to its old value V **a special log record <T_i , X, V> is written out**
    - When undo of a transaction is complete, a log record <T_i **abort**> is written out
      - To indicate that the undo has completed
  - **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
    - No logging is done in this case

Undo and Redo on Recovering from Failure

- When recovering after failure
  - To ensure atomicity, **system consults the log** to determine which **transactions need to be REDONE** and which **need to be UNDONE**
  - Transaction T_i needs to be **UNDONE** if the log
    - Contains the record <T_i **start**>,
    - But does not contain either the record <T_i **commit**> *or* <T_i **abort**>
  - Transaction T_i needs to be **REDONE** if the log
    - Contains the records <T_i **start**>
    - And contains the record <T_i **commit**> *or* <T_i **abort**>

Immediate DB Modification Recovery Example

The same log as it appears at three instances of time

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>

Recovery actions are:

- Crash occurs just after the log record for the step **write (B)** of T_0
  - System finds the record <T_0 Start> in the log, but no corresponding <T_0 Commit> or <T_0 Abort> record
  - So, transaction T_0 must be undone

Undo (T_0): B is restored to 2000 and A to 1000, and also writes

Log records: <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**>

Immediate DB Modification Recovery Example

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>

- Crash comes just after the log record for the step **write (C)** of T_1 has been written to stable storage
  - **Two recovery actions**
  1. **undo (T_1)** *:* System finds the record <T_1 Start> in the log, but no corresponding <T_1 Commit> or <T_1 Abort> record. So, transaction T_1 must be undone
     - **C is restored to 700** and log records <T_1, C, 700>, <T_0, **abort**> are written out
  2. **redo (T_0)** : System finds the record <T_0 Start> and <T_0 Commit> in the log,
     - ***A* and *B* are set to 950 and 2050**

Immediate DB Modification Recovery Example

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

Two Recovery actions are:

- The crash occurs just after the log record <T_1 Commit> has been written to stable storage
  1. **redo (T_1)** *:* System finds the record <T_1 Start> and <T_1 Commit>
     - **C is restored to 600**
  2. **redo (T_0)** : System finds the record <T_0 Start> and <T_0 Commit>
     - ***A* and *B* are set to 950 and 2050**

Undo and Redo on Recovering from Failure

- Note that If transaction T_i was undone earlier and the <T_i **abort**> record written to the log, and then a failure occurs, on recovery from failure T_i is redone
  - **Such a redo redoes all the original actions *including the steps that restored old values***
    - Known as **repeating history**
    - Seems wasteful, but simplifies recovery greatly

Checkpoints

- **Redoing/undoing** all transactions recorded in the log can be **very slow**
  1. Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system
  2. As time passes, the log file may grow too big to be handled at all
  3. Processing the entire log is time-consuming if the system has run for a long time
  4. **Unnecessarily redo transactions which have already output their updates to the database**

Checkpoints

- Checkpoint
  - A mechanism where all the previous logs are removed from the system and stored permanently in a storage disk
  - Declares a point before which the DBMS was in consistent state, and all the transactions were committed

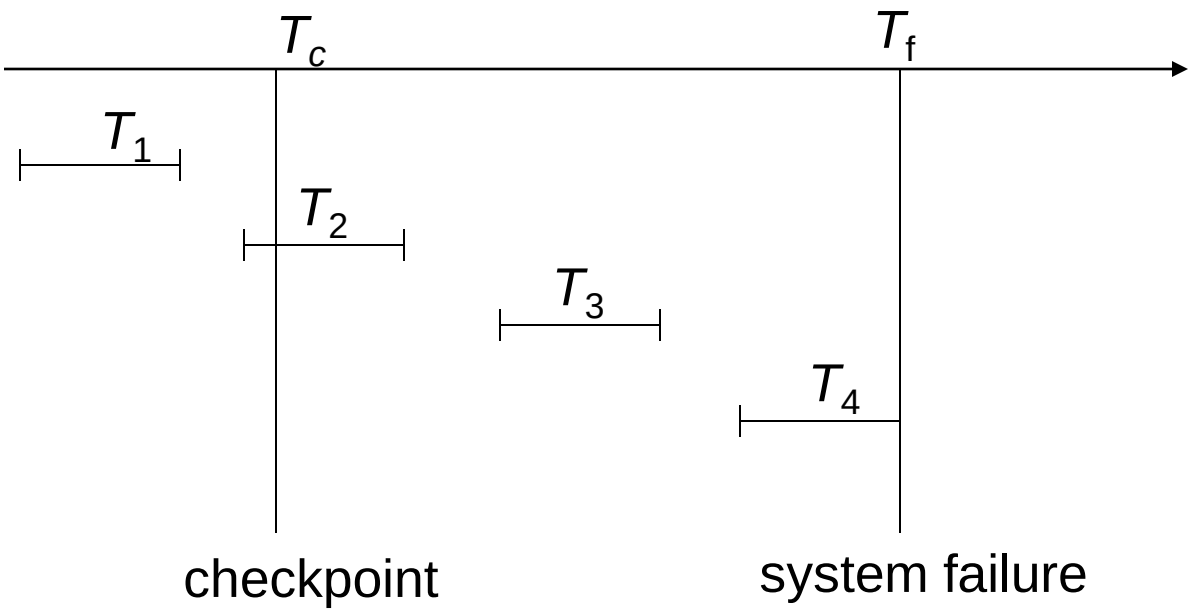
Checkpoints

- 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** *L* > onto stable storage where *L* is a list of all transactions active at the time of checkpoint
  - All updates are stopped while doing checkpointing

Checkpoints (Cont.)

- **During recovery, need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i**
  1. Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
  2. Only transactions that are in *L* or started after the checkpoint need to be redone or undone
     - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage
  3. Some earlier part of the log may be needed for undo operations
  4. Continue scanning backwards till a record <T_i **start**> is found for every transaction T_i in *L*
     - Parts of log prior to earliest <T_i **start**> record above are not needed for recovery, and can be erased whenever desired

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

Recovery Algorithm

- **So far:** covered key concepts
- **Now**: present the components of the basic recovery algorithm

Recovery Algorithm

- **Logging** (during normal operation):
  - <T_i **start**> at transaction start
  - <T_i, X_j, V_1, V_2> for each update, and
  - <T_i **commit**> at transaction end
- **Transaction rollback (during normal operation)**
  - Let T_i be the transaction to be rolled back
  - Scan log backwards from the end, and for each log record of T_i of the form <T_i, X_j, V_1, V_2>
    - Perform the undo by writing V_1 to X_j,
    - Write a log record <T_i , X_j, V_1>
      - such log records are called **compensation log records**
  - Once the record <T_i **start**> is found stop the scan and write the log record <T_i **abort**>

Recovery Algorithm (Cont.)

- **Recovery from failure**: Two phases- Redo phase and Undo Phase
- **Redo phase**
  - The system replays updates of **all** transactions *by scanning* the log forward from the last checkpoint, whether they committed, aborted
  - This phase also determines all transactions that were incomplete at the time of the crash, and must therefore be rolled back
    - Such incomplete transactions would either have been active at the time of the checkpoint, and thus would appear in the transaction list in the checkpoint record, or would have started later; further, such incomplete transactions would have neither a $<T_i$ *abort*> nor a $<T_i$ *commit*> *record in the log*
- **At the end of the redo phase, undo-list** contains the list of all transactions that are **incomplete**, that is, they neither committed nor completed rollback before the crash

Recovery Algorithm (Cont.)

- **Undo phase**: undo all incomplete transactions
  - System rolls back all transactions in the undo-list
  - Performs rollback by scanning the log backward from the end

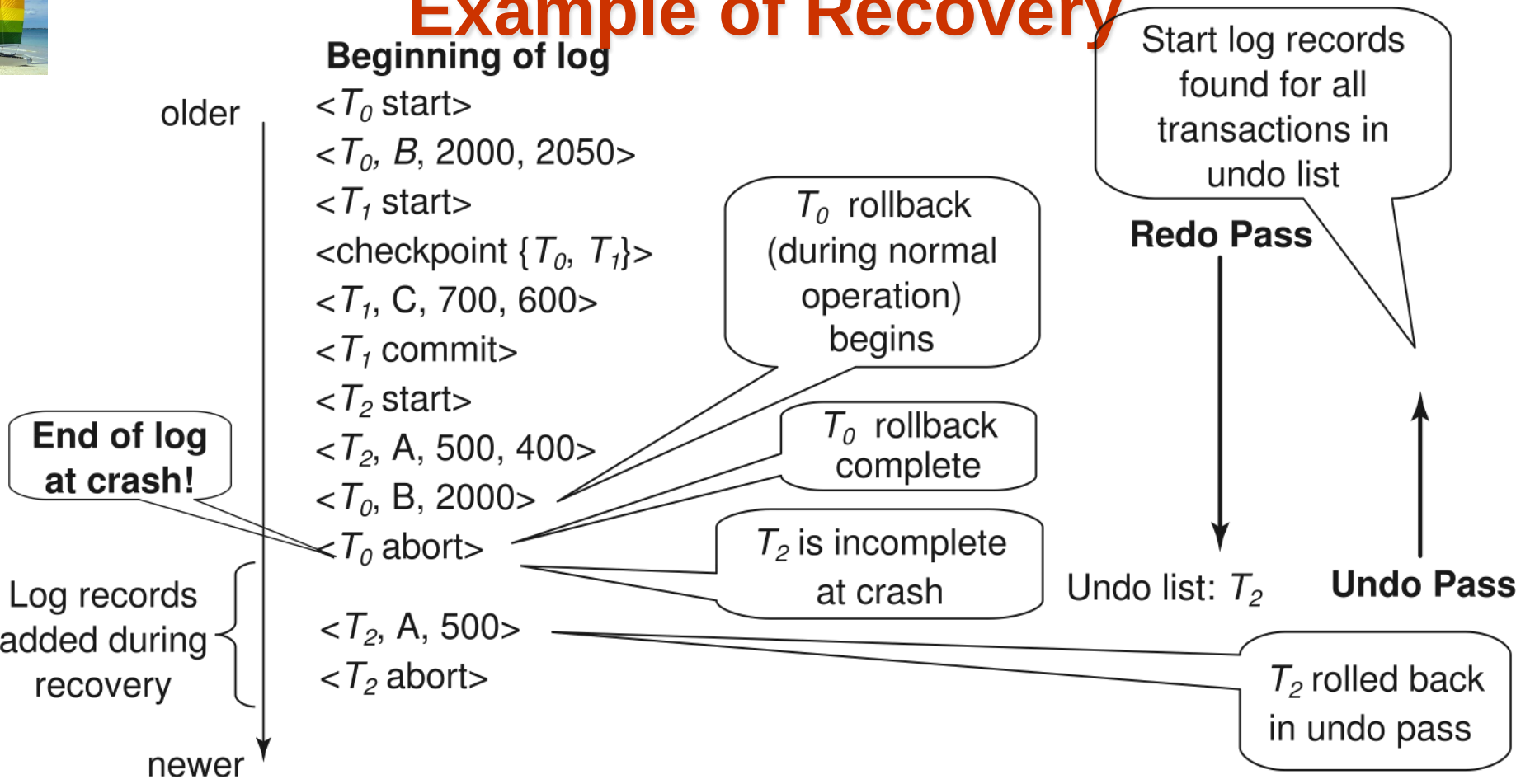
Recovery Algorithm (Cont.)

- **Redo phase**:
  1. Find last <**checkpoint** *L*> record, and set undo-list to *L*
  2. Scan forward from above <**checkpoint** *L*> record
     1. Whenever a record <T_i, X_j, V_1, V_2> is found, redo it by writing V_2 to X_j
     2. Whenever a log record <T_i **start**> is found, add T_i to undo-list
     3. Whenever a log record <T_i **commit**> *or* <T_i **abort**> is found, remove T_i from undo-list

Recovery Algorithm (Cont.)

- **Undo phase:**
  1. Scan log backwards from end
     1. Whenever a log record <T_i, X_j, V_1, V_2> is found where T_i is in undo-list perform same actions as for transaction rollback:
        1. perform undo by writing V_1 to X_j.
        2. write a log record <T_i , X_j, V_1>
     2. Whenever a log record <T_i **start**> is found where T_i is in undo-list,
        1. Write a log record <T_i **abort**>
        2. Remove T_i from undo-list
     3. Stop when undo-list is empty
        - i.e. <T_i **start**> has been found for every transaction in undo-list
- After undo phase completes, normal transaction processing can commence

Example of Recovery

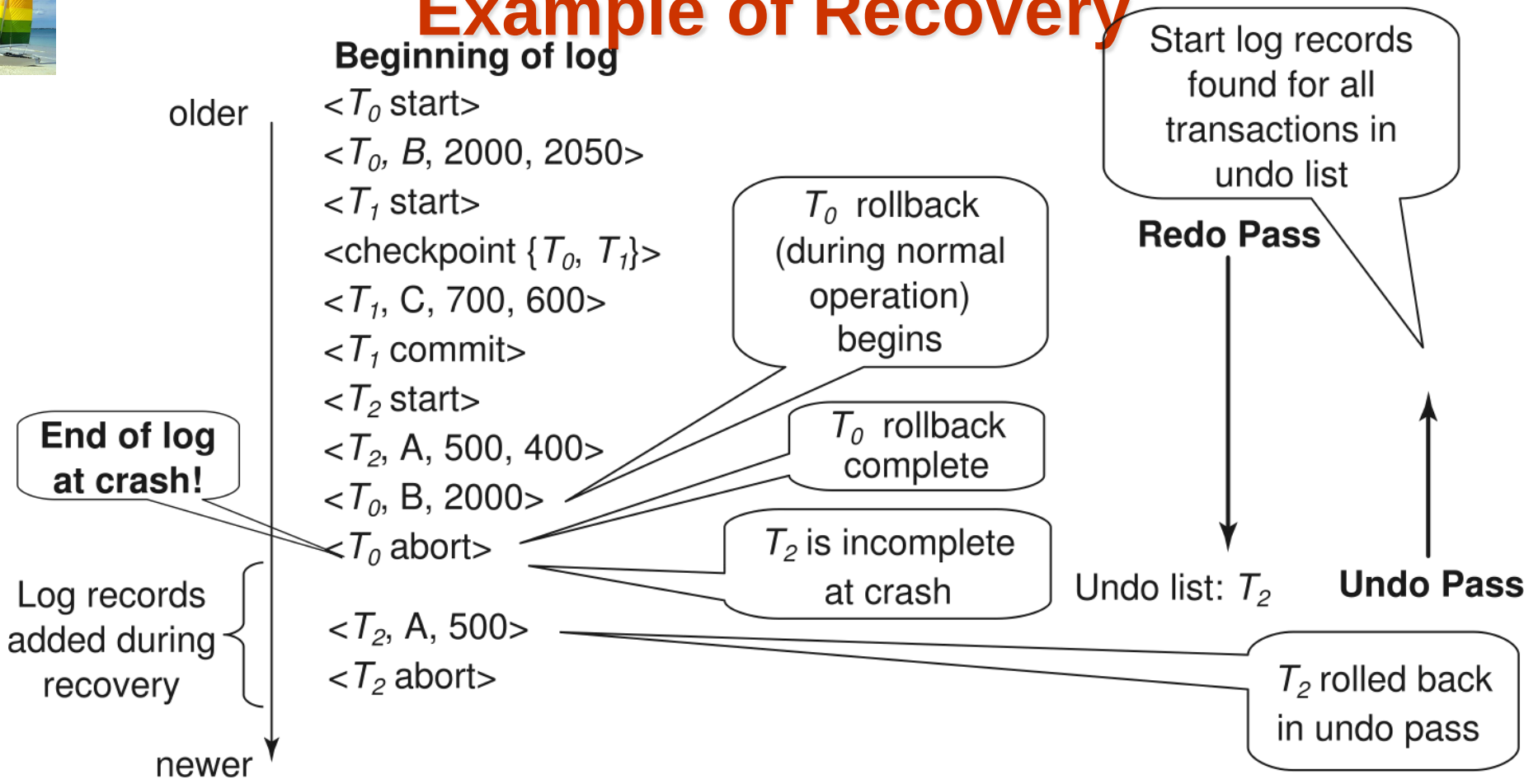


Actions logged during normal operation, and actions performed during failure recovery

Redo phase: System performs a redo of all operations after the last checkpoint record

- undo-list initially contains *T0 and T1*
- *T1 is removed first when its commit log record is* found,
- while *T2 is added when its start log record is found*
- Transaction *T0 is* removed from undo-list when its abort log record is found

Example of Recovery



Actions logged during normal operation, and actions performed during failure recovery

Undo phase: Scans the log backwards from the end, and when it finds

- A log record of *T2 updating A, the old value of A is restored, and a redo-only* log record written to the log
- When the start record for *T2 is found, an abort record* is added for *T2*
- *Since undo-list contains no more transactions, the undo phase* terminates, completing recovery

End

Log Record Buffering

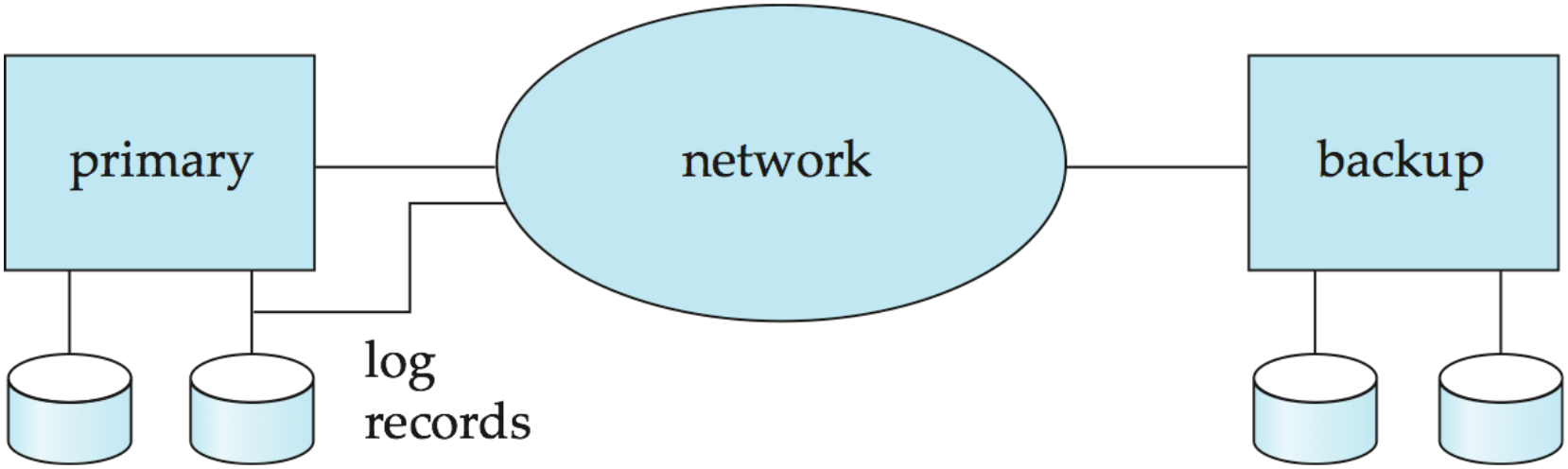
- **Log record buffering**: log records are buffered in main memory, instead of being output directly to stable storage
  - Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed
    - Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage
- Several log records can thus be output using a single output operation, reducing the I/O cost

Log Record Buffering (Cont.)

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created
  - Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage
  - 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
    - This rule is called the **write-ahead logging** or **WAL** rule
      - Strictly speaking WAL only requires undo information to be output

Remote Backup Systems

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.



Remote Backup Systems (Cont.)

- **Detection of failure**: Backup site must detect when primary site has failed
  - to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.
  - Heart-beat messages
- **Transfer of control**:
  - To take over control backup site first perform recovery using its copy of the database and all the long records it has received from the primary.
    - Thus, completed transactions are redone and incomplete transactions are rolled back.
  - When the backup site takes over processing it becomes the new primary
  - To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.

Remote Backup Systems (Cont.)

- **Time to recover**: To reduce delay in takeover, backup site periodically proceses the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.
- **Hot-Spare** configuration permits very fast takeover:
  - Backup continually processes redo log record as they arrive, applying the updates locally.
  - When failure of the primary is detected the backup rolls back incomplete transactions, and is ready to process new transactions.
- Alternative to remote backup: distributed database with replicated data
  - Remote backup is faster and cheaper, but less tolerant to failure
    - more on this in Chapter 19

Remote Backup Systems (Cont.)

- Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.
- **One-safe:** commit as soon as transaction's commit log record is written at primary
  - Problem: updates may not arrive at backup before it takes over.
- **Two-very-safe:** commit when transaction's commit log record is written at primary and backup
  - Reduces availability since transactions cannot commit if either site fails.
- **Two-safe:** proceed as in two-very-safe if both primary and backup are active. If only the primary is active, the transaction commits as soon as is commit log record is written at the primary.
  - Better availability than two-very-safe; avoids problem of lost transactions in one-safe.

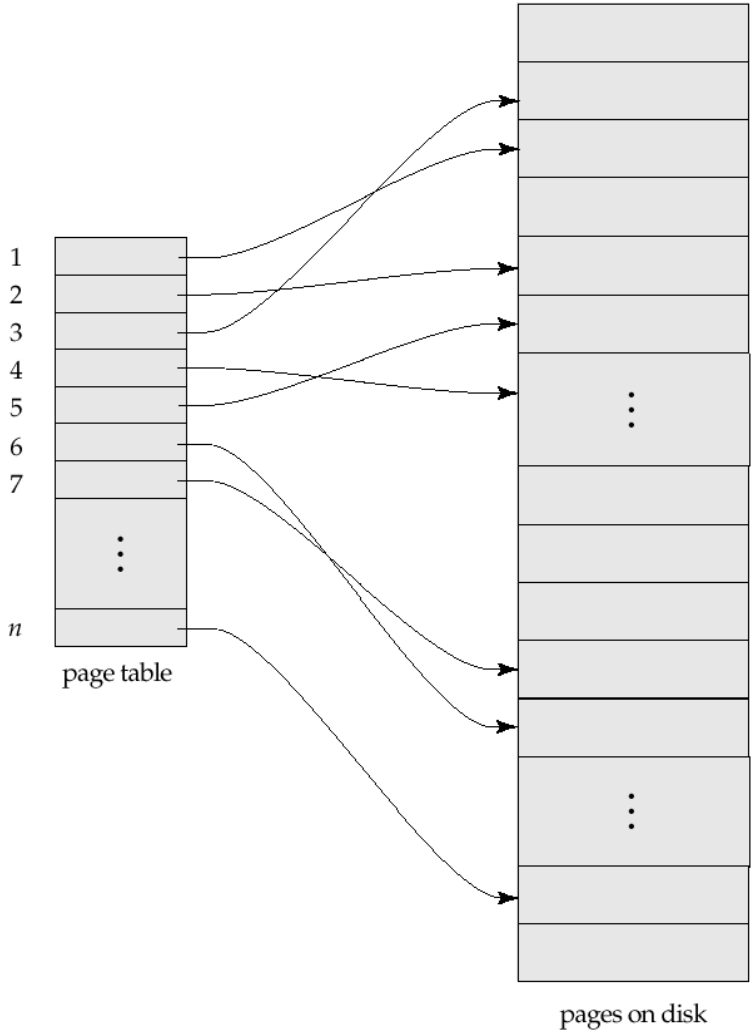
End of Chapter 16

Extra

Shadow Paging

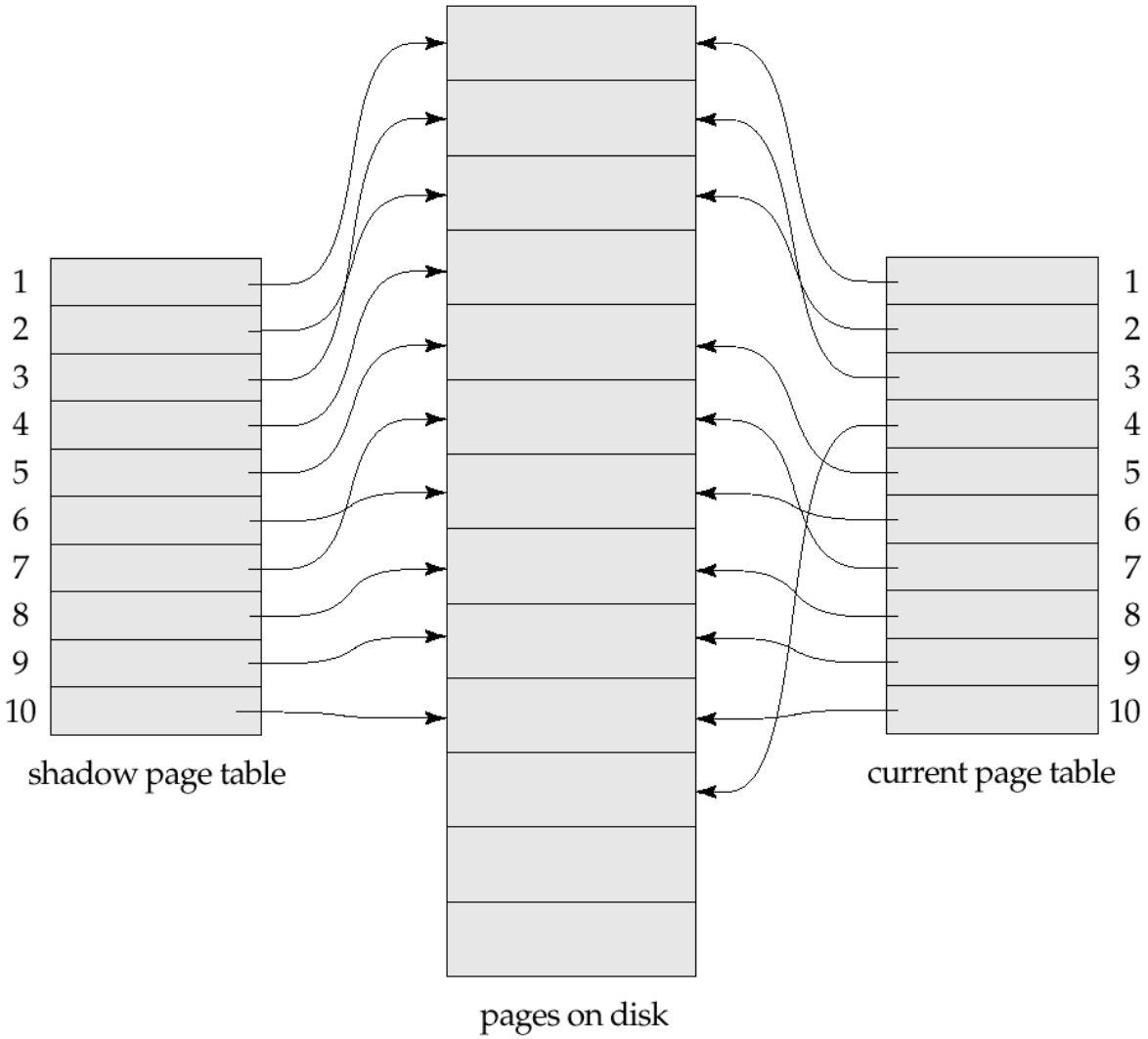
- **Shadow paging** is an alternative to log-based recovery; this scheme is useful if transactions execute serially
- Idea: maintain *two* page tables during the lifetime of a transaction –the **current page table**, and the **shadow page table**
- Store the shadow page table in nonvolatile storage, 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

Sample Page Table



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).

Shadow 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 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