Recovery System

Dr. Shrutilipi Bhattacharjee, Assistant Professor, Dept. of IT, NIT Karnataka, India

Database System Recovery

- All database reads/writes are within a transaction

- Transactions have the "ACID" properties
  - Atomicity: All or nothing
  - Consistency: Preserves database integrity
  - Isolation: Execute as if they were run alone
  - Durability: Results are not lost by a failure

- Concurrency control guarantees I, contributes to C

- Application program guarantees C

- Recovery subsystem guarantees A & D, contributes to C

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 be 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

- Suppose transaction T_i 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 have two parts:
  - Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  - 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

- **Non-volatile storage**:
  - Survives system crashes
  - Examples: Disk, tape, flash memory, non-volatile (battery backed up) RAM
  - But may still fail, losing data

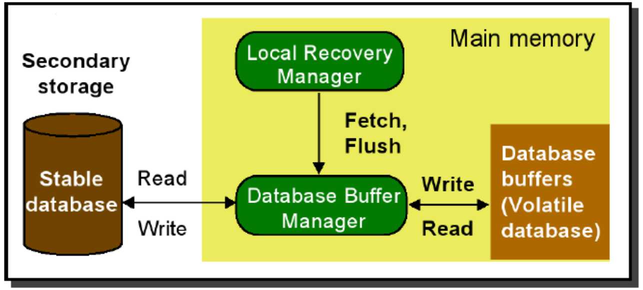
- **Stable storage**:
  - A mythical form of storage that survives all failures
  - Approximated by maintaining multiple copies on distinct nonvolatile media

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. 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):
    - Write the information onto the first physical block
    - When the first write successfully completes, write the same information onto the second physical block
    - The output is completed only after the second write successfully completes



Stable-Storage Implementation

Protecting storage media from failure during data transfer

- Copies of a block may differ due to failure during output operation

- To recover from failure:
  - First find inconsistent blocks:
    - *Expensive solution*: Compare the two copies of every disk block
    - *Better solution*:
      - Record in-progress disk writes on non-volatile storage (Flash, 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

  - 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

Data Access

- **Physical blocks** are those blocks residing on the disk

- **Buffer blocks** are the blocks residing temporarily in main memory

- Block movements between disk and main memory are initiated through the following 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

- We assume, for simplicity, that each data item fits in, and is stored inside, a single block

Data Access

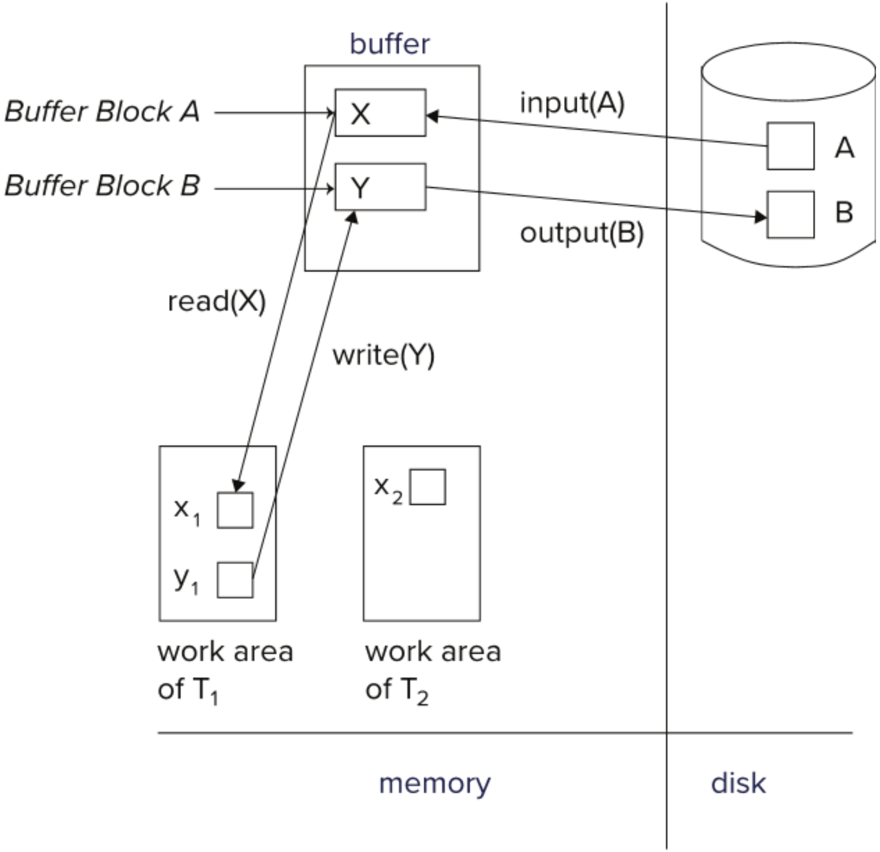
- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept
  - T_i's local copy of a data item X is called x_i
  - B_X denotes block containing X

- 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

- 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

- Note that **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

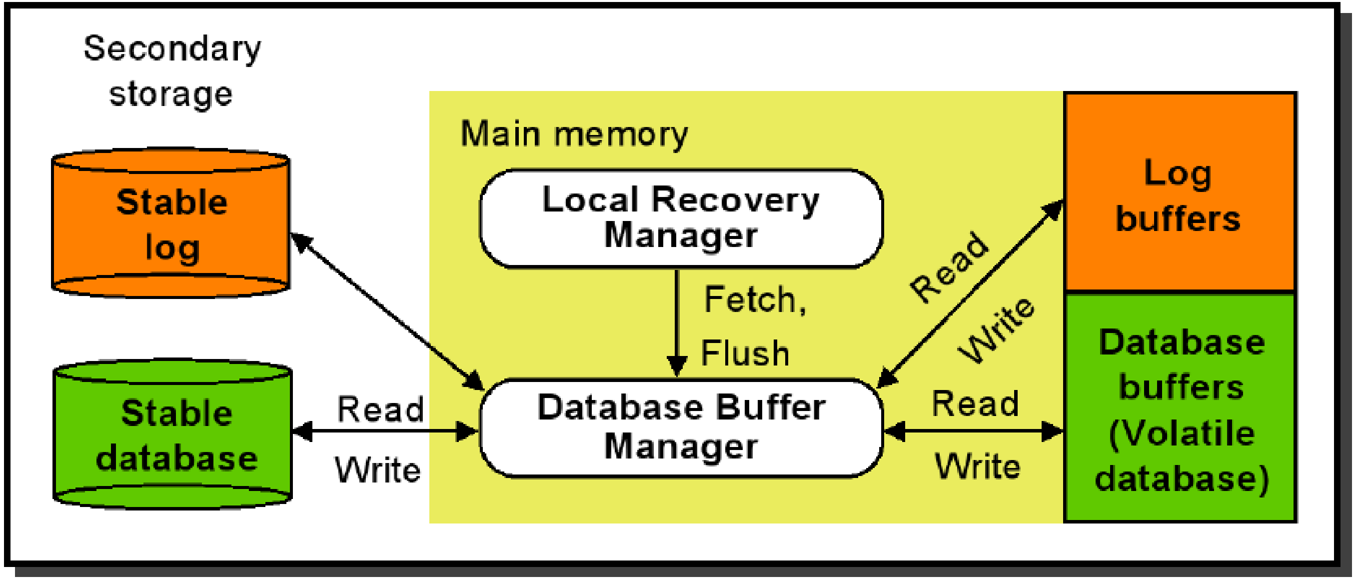
- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself

- We study **log-based recovery mechanisms** in detail
  - We first present key concepts
  - And then present the actual recovery algorithm

- Less used alternative: **shadow-copy** and **shadow-paging**

- In this module, we assume serial execution of transactions

Log-based Recovery



Log-based Recovery

- A **log** is a sequence of **log records** which keep information about update activities on the database
- The **log** is kept on stable storage
- When transaction T_i starts, it registers itself by writing a record to the log

  <T_i **start**>

- *Before* T_i executes **write**(X), a log record is written, where V_1 is the value of X before the write (the **old value**), and V_2 is the value to be written to X (the **new value**)

  $<T_i, X, V_1, V_2>$

- When T_i finishes it last statement, the log record <T_i **commit**> is written
- Two approaches using logs
  - Immediate database modification
  - Deferred database modification

Database Modification

- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
- Update log record must be written ***before*** a database item is written
  - We assume that the log record is output directly to stable storage
- Output of updated blocks to disk storage 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
- The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit
  - Simplifies some aspects of recovery
  - But has overhead of storing local copy
- We cover here only the immediate-modification scheme

Transaction Commit

- A transaction is said to have committed when its commit log record is output to stable storage
  - All previous log records of the transaction must have been output already
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

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

B_C output before T_1 commits

- Note: B_X denotes block containing X

B_A output after T_0 commits

Undo and Redo Operations

- What is the use of the logs?

- **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 (called **redo-only**) < 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 was 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 Operations

- The **undo** and **redo** operations are used in several different circumstances:
  - The **undo is** used for transaction rollback during normal operation
    - In case a transaction cannot complete its execution due to some logical error
  - The **undo** and **redo** operations are used during recovery from failure
- We need to deal with the case where during recovery from failure another failure occurs prior to the system having fully recovered

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**>

Undo and Redo on Recovering from Failure

- When recovering after failure:
  - 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**>

  - It may seem strange to redo transaction T_i if the record <T_i **abort** > record is in the log
    - To see why this works, note that if <T_i **abort** > is in the log, so are the redo-only records written by the undo operation
    - Thus, the end result will be to undo T_i's modifications in this case
    - This slight redundancy simplifies the recovery algorithm and enables faster overall recovery time
    - Such a redo redoes all the original actions including the steps that restored old value - Known as **repeating history**

Immediate 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, and log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> are written out
  - (b) redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and C is restored to 700. Log records <T_1, C, 700>, <T_1, **abort**> are written out
  - (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

- Redoing/undoing all transactions recorded in the log can be very slow
  - Processing the entire log is time-consuming if the system has run for a long time
  - We might unnecessarily redo transactions which have already output their updates to the database

- Streamline recovery procedure by periodically performing **checkpointing**

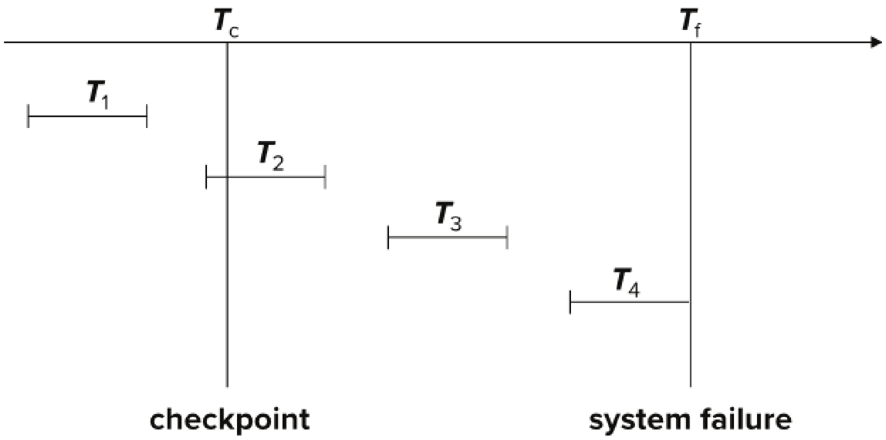
- All updates are stopped while doing checkpointing
  - Output all log records currently residing in main memory onto stable storage
  - Output all modified buffer blocks to the disk
  - Write a log record < **checkpoint** *L* > onto stable storage where *L* is a list of all transactions active at the time of checkpoint

Checkpoints

- 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** L > record
  - 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

- Some earlier part of the log may be needed for undo operations
  - 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



- Any transactions that committed before the last checkpoint should be ignored
- T_1 can be ignored (updates already output to disk due to checkpoint)

- Any transactions that committed since the last checkpoint need to be redone
- T_2 and T_3 redone

- Any transaction that was running at the time of failure needs to be undone and restarted
- T_4 undone

Next Lecture

Data Analytics: Data Warehousing, Data Mining

Thank you for your attention...

Any question?

Contact:
Department of Information Technology, NITK Surathkal, India
6th Floor, Room: 13
Phone: +91-9477678768
E-mail: shrutilipi@nitk.edu.in