Chapter 15: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Shadow Paging
- Recovery With Concurrent Transactions
- Buffer Management
- Failure with Loss of Nonvolatile Storage
- Advanced Recovery Techniques

Failure Classification

- 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 failure causes the system to crash
- Disk failure: a head crash of similar failure destroys all or part of disk storage

Storage Structure

- Volatile storage:
  - does not survive system crashes
  - examples: main memory, cache memory
- Nonvolatile storage:
  - survives system crashes
  - examples: disk, tape
- Stable storage:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media

Stable-Storage Implementation

- Protect storage media from failure during data transfer; system must maintain two physical blocks for each logical database block.
- Execute output operation as follows:

  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.

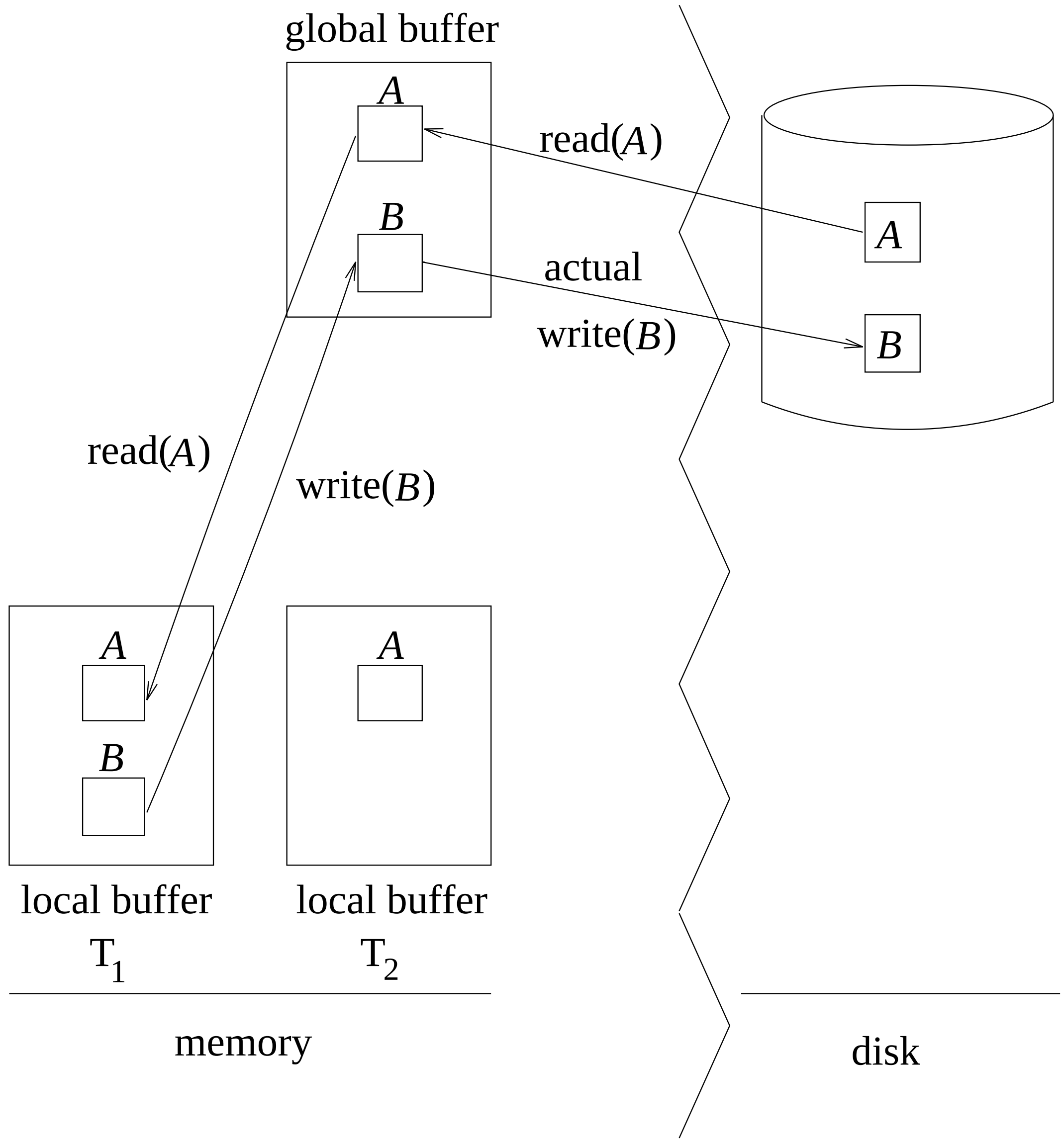
Data Access

- *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.

Data Transfer

- The transfer of data between the database and program variables is accomplished using:
  - **read**(X) assigns the value of data item X to the local variable x_i.
    - If the block B_X on which X resides is not in main memory, then issue **input**(B_X).
    - Assign to x_i the value of X from the buffer block.
  - **write**(X) assigns the value of local variable x_i to data item X in the buffer block.
    - If block B_X is not in main memory, then issue **input**(B_X).
    - Assign the value of x_i to X in buffer B_X.

Example of Data Transfer and Access



Assumptions

- Each data item can be read and written only once by one single transaction. It can be modified many times in the local buffer
- Values are written onto the global buffer space in the same sequential order that the write instructions are issued
- If the transaction reads a value from the data base and this value was previously modified, it always gets the latest value
- Comment: a data "item" can be a file, relation, record, physical page, etc. The choice is up to the designer.

Recovery and Atomicity

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- For 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.
- If T_i performed multiple database modifications, several output operations may be required and a failure may occur after some of these modifications have been made but before all of them are made.
- To ensure atomicity, first output information describing the modifications to stable storage without modifying the database itself.

Log-Based Recovery

- A log file is kept on stable storage
- When transaction T_i starts, it registers itself on the log by writing $<T_i$ **start**$>$
- Whenever T_i executes write(X), the fields
  - transaction name (i.e., T_i)
  - data item name (i.e., X)
  - old value (e.g., V_1)
  - new value (e.g., V_2)

  are written sequentially on the log, and then the write(X) is executed

Log-Based Recovery (Cont.)

- When T_i reaches it last statement, the record <T_i **commit**> is added to the log
- If X is modified then its corresponding log record is always first actually written on the log (stable storage) and then actually written on the database
- Before T_i is committed, all its corresponding log records must be in stable storage

Example of Recovery

- Consider transactions T_1 and T_2 which are executed sequentially by the system, and with initial values of $A = 100$, $B = 300$, $C = 5$, $D = 60$, $E = 80$

T_1:
read(A)
$A := A + 50$
read(B)
$B := B + 100$
write(B)
read(C)
$C := 2C$
write(C)
$A := A + B + C$
write(A)

T_2:
read(A)
$A := A + 10$
write(A)
read(D)
$D := D - 10$
read(E)
read(B)
$E := E + B$
write(E)
$D := D + E$
write(D)

Example of Recovery (Cont.)

Log records	
1.	$<T_1$ **start**$>$
2.	$<T_1$, old B: 300, new B: 400$>$
3.	$<T_1$, old C: 5, new C: 10$>$
4.	$<T_1$, old A: 100, new A: 560$>$
5.	$<T_1$ **commit**$>$
6.	$<T_2$ **start**$>$
7.	$<T_2$, old A: 560, new A: 570$>$
8.	$<T_2$, old E: 80, new E: 480$>$
9.	$<T_2$, old D: 60, new D: 530$>$
10.	$<T_2$ **commit**$>$

Database values		
I.	B	400
II.	C	10
III.	A	560
IV.	A	570
V.	E	480
VI.	D	530

Example of Recovery (Cont.)

The order of actual writes to log and database might be:

Time	Log	Database
	1	
	2	
	3	
	4	
	5	
		I
		II
		III
	6	
	7	
	8	
		IV
	9	
	10	
		V
		VI

Example of Recovery (Cont.)

If a crash occurs, the log is examined and various actions are taken depending on the last instruction (actually) written on it.

Last instruction (I)	Action
I = 0	nothing
$1 \leq I \leq 4$	undo(T_1): restore the values of the variables modified by T_1 to old values Consequence: T_1 has not run
$5 \leq I \leq 9$	redo(T_1): set the values of the variables modified by T_1 to the values created by T_1 undo(T_2): Consequence: T_1 ran, T_2 has not run
I = 10	redo(T_1): redo(T_2): Consequence: T_1, T_2 ran

The Algorithm

- Redo all transactions for which the log has both **start** and **commit**
- Undo all transactions for which the log has **start** but no **commit**
- Remarks:
  - In a multitasking system more than one transaction may need to be undone
  - If a system crashes during the recovery stage, the new recovery must still give correct results
  - In this algorithm, a large number of transactions need to be redone, since we do not know how far behind the database updates are

Checkpoints

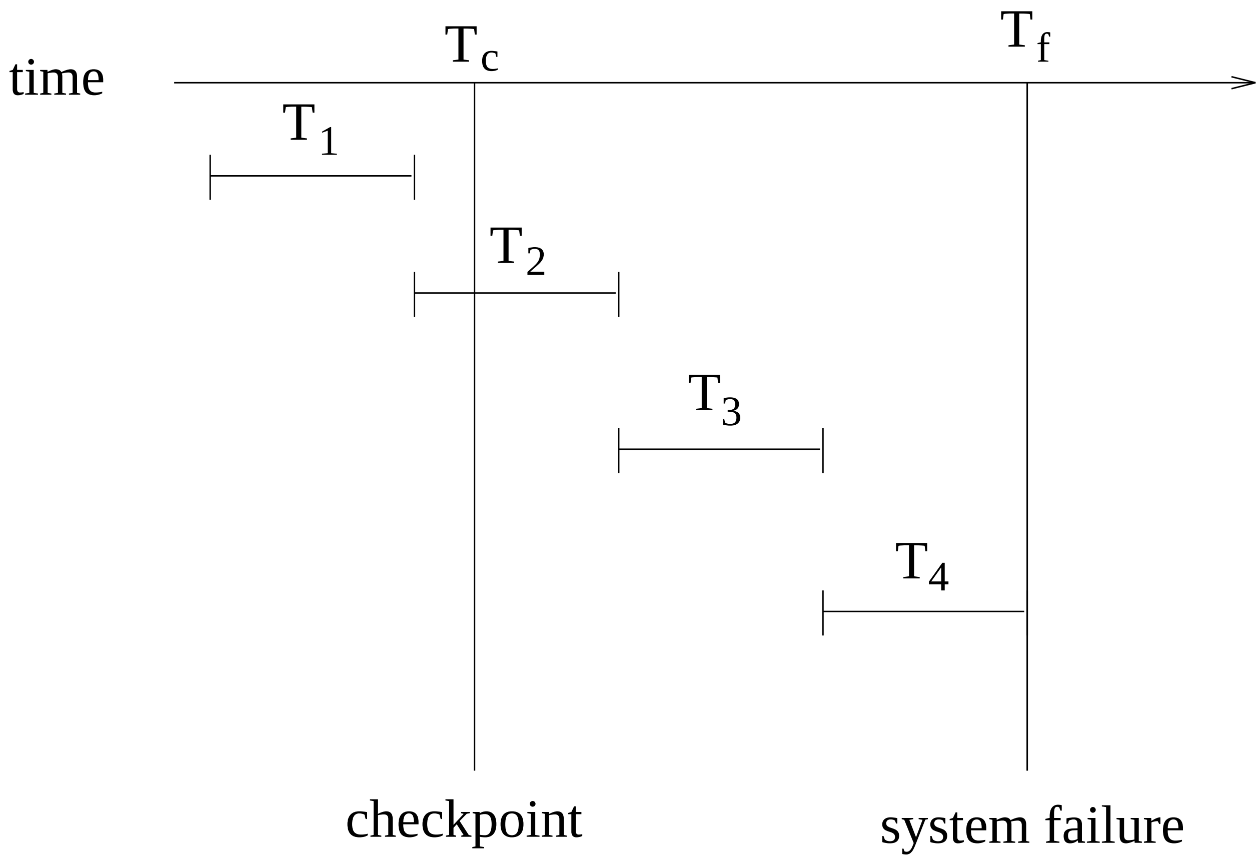
- Streamline recovery procedure by periodically performing *checkpointing*

  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Output a log record <**checkpoint**> onto stable storage.

- During recovery

  – Undo all transactions that have not committed.
  – Redo all transactions that have committed after checkpoint.

Example of Checkpoints



- T_1 ok
- T_2 and T_3 redone
- T_4 undone

Shadow Paging

- Alternative to log-based recovery; maintain *two* page tables during the life of a transaction
  - *current* page table
  - *shadow* page table
- Store the shadow page table in nonvolatile storage; state of the database prior to transaction execution may be recovered.

Shadow Paging (Cont.)

- When the transaction commits, the current page table is written to nonvolatile storage.
- The current page table then becomes the new shadow page table and the next transaction is allowed to begin execution.
- Eliminates overhead of log-record output; recovery from crashes is faster.
- Drawbacks:
  - data fragmentation
  - garbage collection

Recovery With Concurrent Transactions

- Log scanning – when transactions execute concurrently, several transactions may have been active at the time of the last checkpoint.
- Concurrent transaction-processing system requires that the checkpoint log record be of the form

  <**checkpoint** *L*>

  where *L* is a list of transactions active at the time of the checkpoint.
- When the system recovers from a crash it constructs two lists:
  - *undo-list* consists of transactions to be undone.
  - *redo-list* consists of transactions to be redone.

Recovery With Concurrent Tran. (Cont.)

- Once the redo-list and undo-list have been constructed, recovery proceeds as follows:

  1. Rescan log from most recent record backward until the <**checkpoint** L> record; perform **undo**(T_i) for each T_i on the *undo-list*.
  2. Continue to scan log backward, performing **undo**(T_i) for each T_i on the *undo-list*, until the $< T_i$ **starts** $>$ record for all T_i on the *undo-list* has been located.
  3. Scan log forward from <**checkpoint** L> record and perform **redo**(T_i) for each T_i on the *redo-list*.

- After all transactions on *undo-list* have been undone, transactions on the *redo-list* are redone.

Buffer Management

- Log record buffering
  - Transaction T_i enters the commit state after the $<T_i$ **commit**$>$ log record has been output to stable storage.
  - Before the $<T_i$ **commit**$>$ log record may be output to stable storage, all log records pertaining to transaction T_i must have 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.

Buffer Management (Cont.)

- Database buffering – Overwrite block B_1 in main memory when another block B_2 needs to be brought into memory; ifB_1 has been modified, B_1 must be output prior to B_2's input (*virtual memory*).
  - Output to stable storage all block B_1's log records.
  - Output block B_1 to disk.
  - Input block B_2 from disk to main memory.
- If the OS cannot enforce output of log records prior to output of database blocks, database cannot utilize virtual memory.
  - DB reserves part of main memory as a buffer and manages data block transfer; limits amount of main memory available to the database buffer.
  - DB implements its buffer within the virtual memory of the operating system; may result in extra output of data to disk.

Failure with Loss of Nonvolatile Storage

- Periodically *dump* the entire content of the database to stable storage
- No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
  - Output all log records currently residing in main memory onto stable storage.
  - Output all buffer blocks onto the disk.
  - Copy the contents of the database to stable storage.
  - Output a log record <**dump**> onto the stable storage.

Advanced Recovery Techniques

- Support high-concurrency locking techniques, such as those used for B^{+}-tree concurrency control; based on logical (operation) undo, and follow the principle of repeating history.
- When recovering from system failure, perform a redo pass using the log, followed by an undo pass on the log to roll back incomplete transactions.
- Logical undo logging
- Transaction rollback
- Checkpoints
- Restart recovery
- Fuzzy checkpointing