Contents

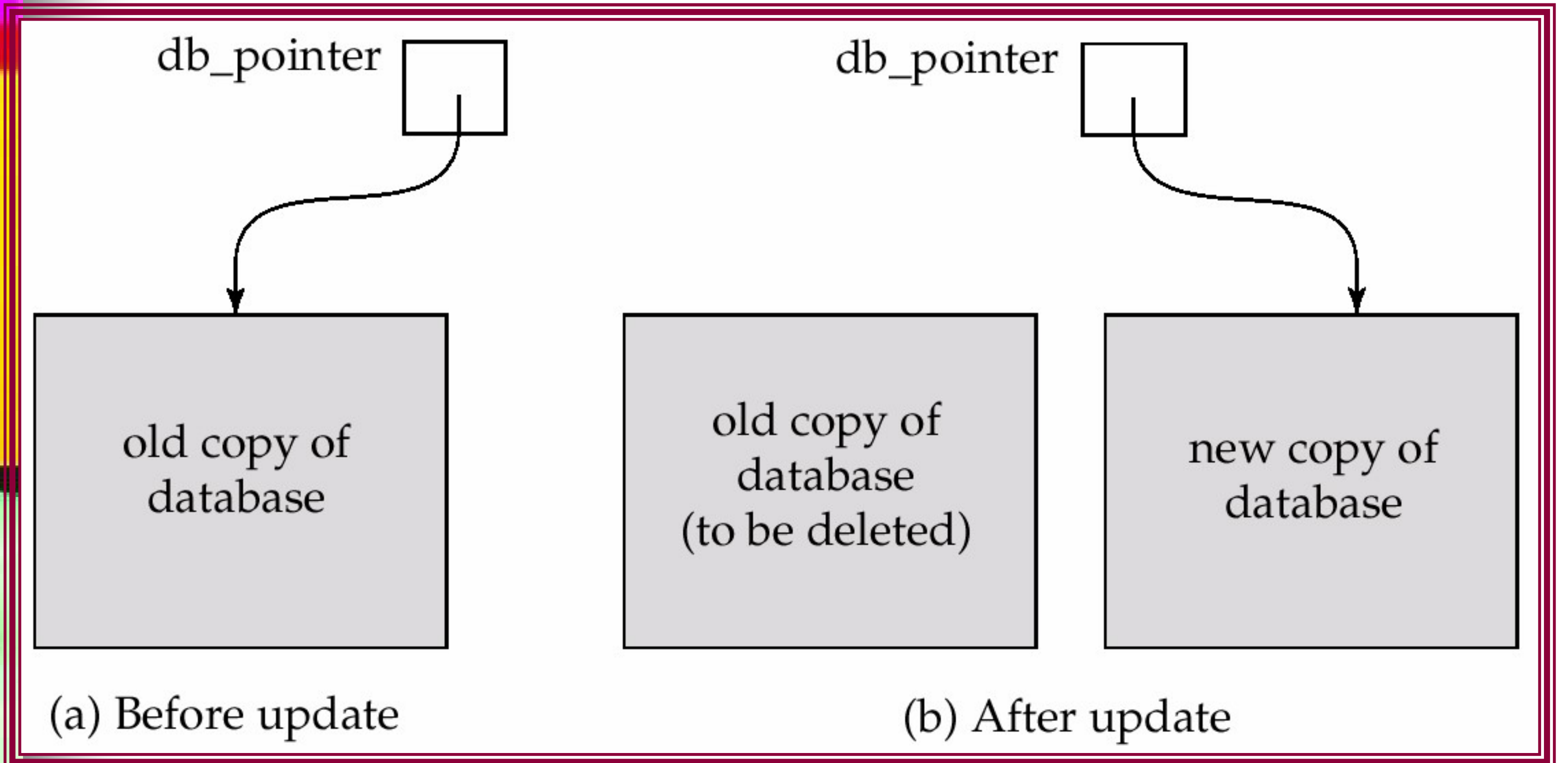
Implementation of Atomicity and Durability

- The recovery-management component of a database system implements the support for atomicity and durability.

- The *shadow-database* scheme:
  - The scheme, which is based on making copies of the database, called shadow copies
  - assume that only one transaction is active at a time.
  - a pointer called db_pointer always points to the current consistent copy of the database.
  - all updates are made on a *shadow copy* of the database, and **db_pointer** is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
  - in case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.

Implementation of Atomicity and Durability (Cont.)

The shadow-database scheme:



- Assumes disks to not fail
- Useful for text editors, but extremely inefficient for large databases: executing a single transaction requires copying the *entire* database.

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- *Concurrency control schemes* – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- *Schedules* – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.

Example Schedules

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B. The following is a serial schedule (Schedule 1 in the text), in which T_1 is followed by T_2.

	T_1	T_2
1	read(A)	
2	$A := A - 50$	
3	write (A)	
4	read(B)	
5	$B := B + 50$	
6	write(B)	
		read(A)
		$temp := A * 0.1$
		$A := A - temp$
		write(A)
		read(B)
		$B := B + temp$
		write(B)

Example Schedule (Cont.)

- Let T_1 and T_2 be the transactions defined previously. The following schedule (Schedule 3 in the text) is not a serial schedule, but it is *equivalent* to Schedule 1.

T_1	T_2
read(A) $A := A - 50$ write(A)	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A)
read(B) $B := B + 50$ write(B)	
	read(B) $B := B + temp$ write(B)

In both Schedule 1 and 3, the sum A + B is preserved.

Example Schedules (Cont.)

- The following concurrent schedule (Schedule 4 in the text) does not preserve the value of the the sum $A + B$.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)