Intro to DB

CHAPTER 15
TRANSACTION MNGMNT

Chapter 15: Transactions

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Testing for Serializability

Transactions

- What is a transaction?
  - Logical unit of work (user view)
  - A program unit that accesses and possibly updates various data items (system view)
  - DBMS must guarantee certain properties (ACID properties) for units of works declared as a DB transaction

Examples of Transactions

- Banks
  - Withdraw $100 to account A.
  - Transfer $50 from account A to B.
- Schools
  - Register course #409.433 for student #4321.
- Airlines
  - Check if two seats are available on flight #453.
  - Reserve the two seats on flight #453.
- Companies
  - Increase every employee's salary by 5%.

Dangers for Transactions

- Various types of failure
  - system crash
  - disk failure
  - system error
- Delayed disk write
  - disk access is performed in chunks: page (block)
  - i.e., write operation performed after the right amount of data has been gathered
  - buffer manager may pin a page

Properties of a Transaction

ACID properties

- **A**tomicity

  "all or nothing"

- **C**onsistency (Correctness)

  Move from a consistent state to another consistent state

- **I**solation

  Should not be interfered by other transactions (concurrency)

- **D**urability

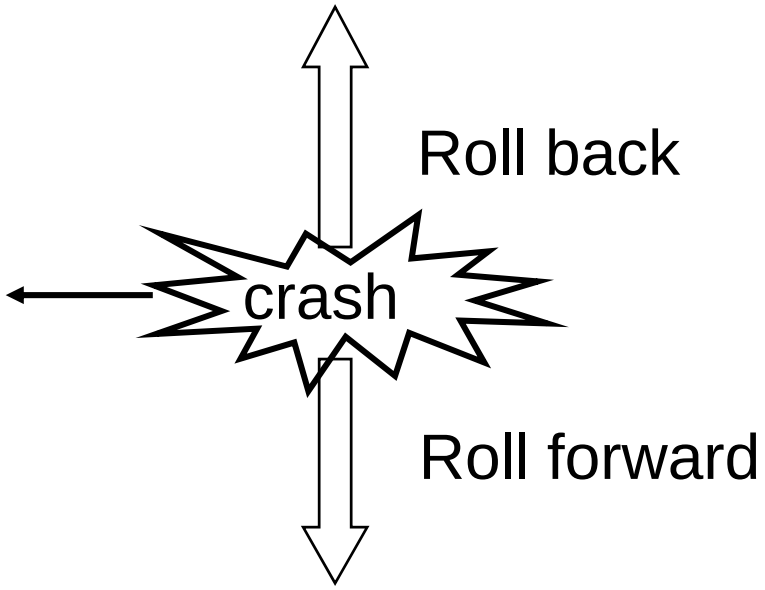
  The effect of a completed transaction should be durable & public

Atomicity

- All or nothing

  "Transfer $50 from account A to account B"

```
Begin transaction
      read(A,a)
      a = a-50
      write(A,a)
      read(B,b)
      b = b+50
      write(B,b)
End Transaction
```



Consistency

- Move from a consistent state to another consistent state

"Withdraw $100 from account A"

```
Begin transaction
        read(A,a)
        a = a-100
        write(A,a)
End Transaction
```

What if A only had $20?

- Responsibility of programmer

Isolation

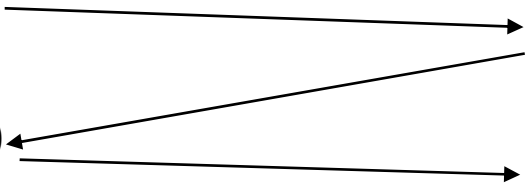
- Should not be interfered by other transactions (concurrency)

"Transaction T1"

```
Begin transaction
  read(A,a1)
  a1 = a1-50
  write(A,a1)
  read(B,b1)
  b1 = b1+50
  write(B,b1)
End Transaction
```

"Transaction T2"

```
Begin transaction
  read(A,a2)
  a2 = a2-100
  write(A,a2)
End Transaction
```



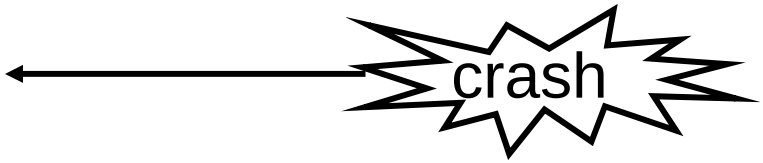
- Serial Execution VS Concurrent Execution

Durability

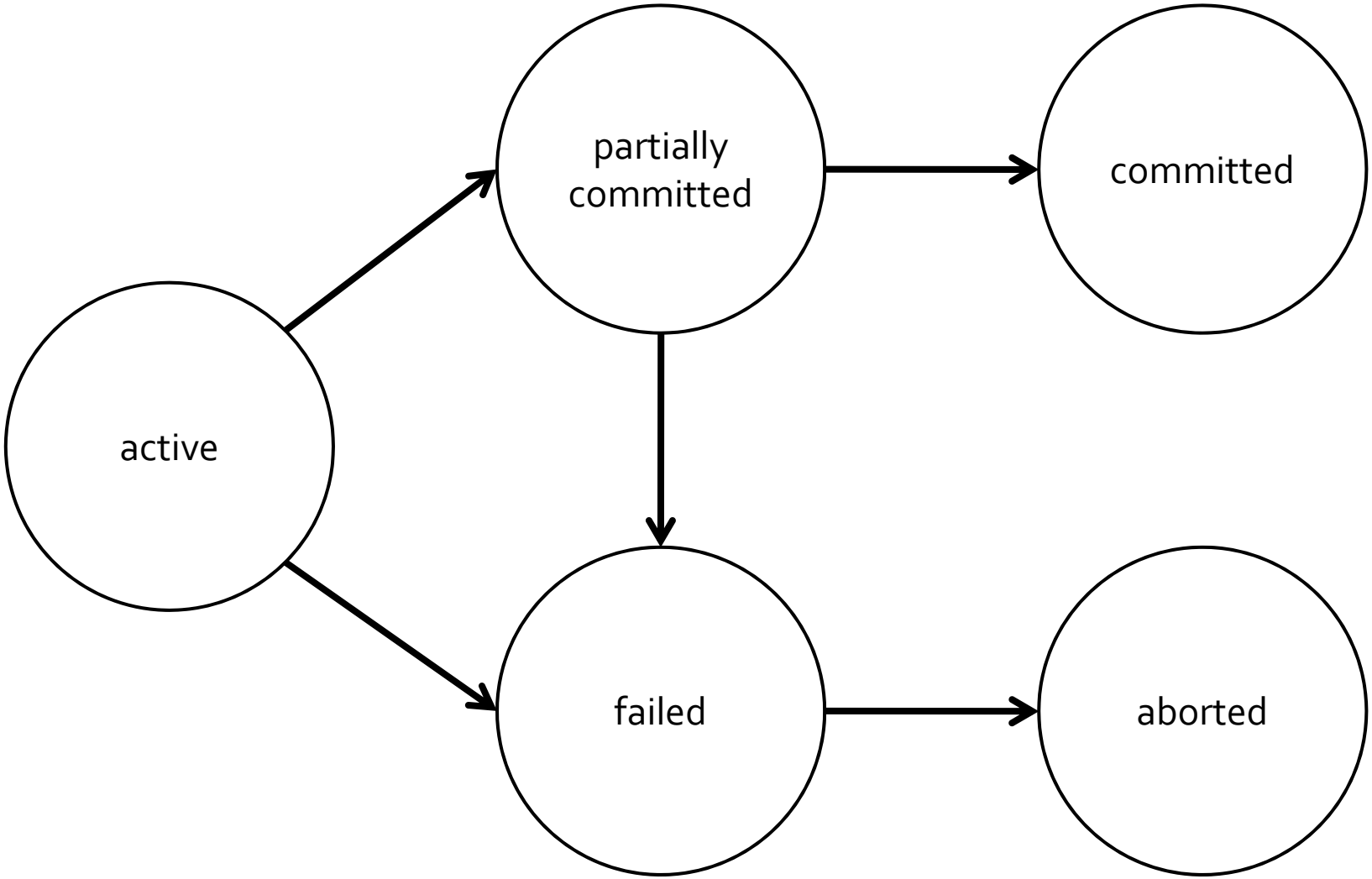
- The effect of a completed transaction should be durable & public

"Withdraw $100 from account A"

```
Begin transaction
      read (A,a)
      a = a-100
      write(A,a)
End Transaction
 ...
*buffer flush*
```



Transaction States



Transaction States (Cont.)

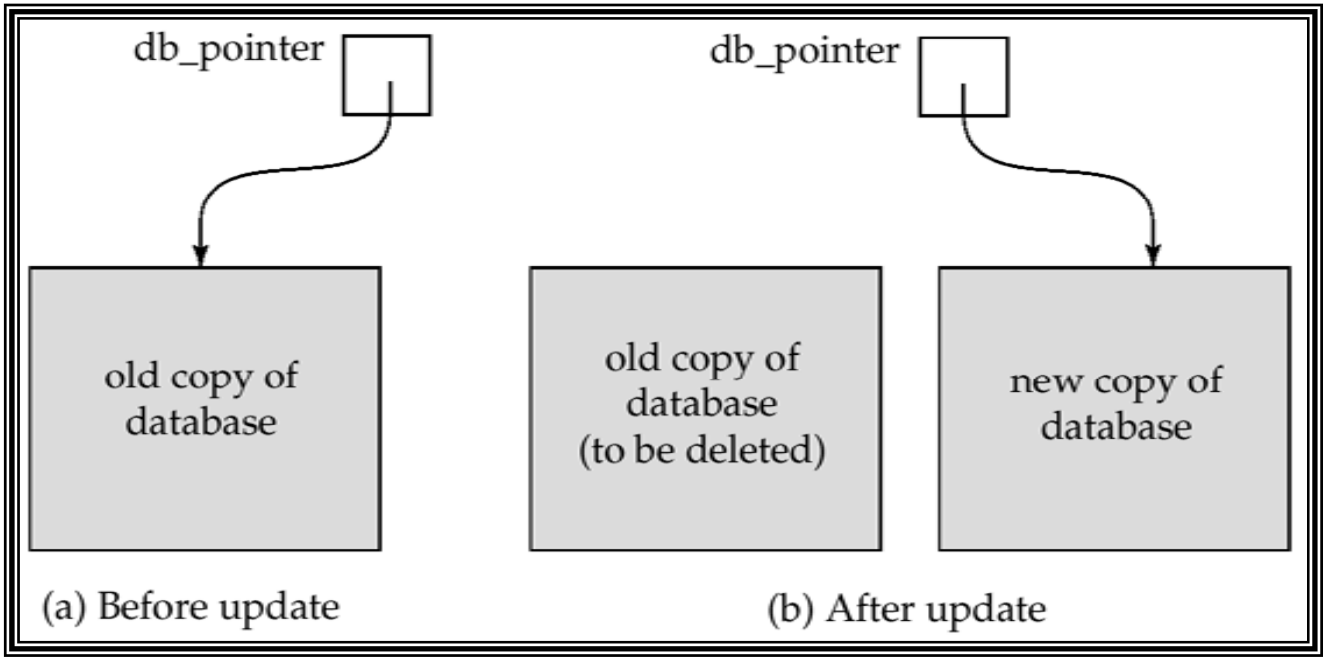
- *Active*: the initial state, transaction stays in this state while it is executing
- *Partially committed*: the final statement has been executed
- *Failed*: normal execution can no longer proceed
- *Aborted*: transaction has been rolled back and the database restored to its state prior to the start of the transaction.
  - Two options after it has been aborted:
    - restart the transaction – only if no internal logical error
    - kill the transaction
- *Committed*: after *successful completion*.

Implementation of Atomicity and Durability

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

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.
- Will see better schemes in Chapter 17. (*Log-based* schemes)



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system
  - increased processor and disk utilization
  - reduced average response time
- *Concurrency control schemes* – mechanisms to achieve isolation
  - to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
  - (Chapter 16)

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 in which T_1 is followed by T_2.

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

Schedule 1 (fig 15.3)

Example Schedule (Cont.)

- Let T_1 and T_2 be the transactions defined previously.
- Schedule 3 is not a serial schedule, but it is equivalent to Sch.1.

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

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)

Schedule 3 (fig 15.5)

Example Schedules (Cont.)

- Schedule 4 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)

Schedule 4 (fig 15.6)

Serializability

- Basic Assumption − Each transaction preserves database consistency.
  - Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.
  1. conflict serializability
  2. view serializability (not covered in this semester)

- We ignore operations other than *read* and *write* instructions.

A Serializable Schedule

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Implementation of Isolation (from Chap 16)

- Schedules must be serializable and recoverable (for database consistency)
  - and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.

Lock-Based Protocols (from Chap 16)

- A lock is a mechanism to control concurrent access to a data item
- Two modes :
  1. *exclusive (X) mode*: both read and write
     (lock-X instruction)
  2. *shared (S) mode*: only read
     (lock-S instruction)
- Lock requests are made to concurrency-control manager
- Transaction can proceed only after request is granted.

Granting of Locks (from Chap 16)

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with lock(s) already held on the item by other transactions
- Any number of transactions can hold shared locks on an item
- If any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

Example (from Chap 16)

T_2: lock-S*(A)*;
read *(A)*;
unlock*(A)*;
lock-S*(B)*;
read *(B)*;
unlock*(B)*;
display*(A+B)*

- Locking as above is not sufficient to guarantee serializability
  — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.

Two-Phase Locking Protocol (from Chap 16)

- Locking Protocol
  - A set of rules followed by all transactions while requesting and releasing locks
  - Locking protocols restrict the set of possible schedules.
- 2PL
  - Phase 1: Growing Phase
    - transaction may obtain locks
      - can acquire a **lock-S** or **lock-X** on item
      - can convert a **lock-S** to a **lock-X** (**upgrade**)
    - transaction may not release locks
  - Phase 2: Shrinking Phase
    - transaction may release locks
      - can release a **lock-S** or **lock-X**
      - can convert a **lock-X** to a **lock-S (downgrade**)
    - transaction may not obtain locks

Example (from Chap 16)

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Features of 2PL (from Chap 16)

- Serializability: the protocol assures (conflict) serializability
  - It can be shown that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock).
  - There can be conflict serializable schedules that cannot be obtained if two-phase locking is used
- Deadlocks: Two-phase locking *does not* ensure freedom from deadlocks
  - starvation also possible
- Cascading rollback: is possible under two-phase locking

Recoverability

- Need to address the effect of transaction failures on concurrently running transactions
- *Recoverable schedule*
  - if a transaction T_j reads a data item previously written by a transaction T_i,
  - the commit of T_i appears before the commit of T_j.
- The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- DBMS must ensure that schedules are recoverable.

Cascading Rollback

- A single transaction failure can lead to a series of transaction rollbacks.
- Example: If T_{10} fails, T_{11} and T_{12} must also be rolled back.

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Can lead to the undoing of a significant amount of work

Cascadeless Schedules

- Cascading rollbacks cannot occur if
  - for each pair of transactions T_i and T_j
  - such that T_j reads a data item previously written by T_i,
  - the commit of T_i appears before the read of T_j

- Every cascadeless schedule is also recoverable

- It is desirable to restrict the schedules to those that are cascadeless

Strict / Rigorous 2PL (from Chap 16)

- Strict 2PL
  - To avoid cascading roll-back
  - A transaction must hold all its exclusive locks until it commits/aborts

- Rigorous 2PL
  - *all* locks are held until commit/abort
  - transactions can be serialized in the order in which they commit

END OF CHAPTER 15