Database Systems
(CS 355 / CE 373)

Dr. Umer Tariq
Assistant Professor,
Dhanani School of Science & Engineering,
Habib University

Acknowledgements

- Many slides have been borrowed from the official lecture slides accompanying the textbook:

  Database System Concepts, (2019), Seventh Edition,

  Avi Silberschatz, Henry F. Korth, S. Sudarshan

  McGraw-Hill, ISBN 9780078022159

  The original lecture slides are available at:

  https://www.db-book.com/

- Some of the slides have been borrowed from the lectures by Dr. Immanuel Trummer (Cornell University). Available at: (www.itrummer.org)

Outline: Week 10

- The Concept of Transaction
- Transaction Properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Transaction States
- Concurrent Execution of Transactions

DBMS: The Concept of *Transactions*

DB (Backend)

Account

Number	C-ID	Balance
1245	13	1000

Account (<u>Number</u>, C-ID, Balance)

Application Code / Program

TransferMy
OnButtonClickedOK()

(SELECT Balance Read(A)
FROM Account
WHERE Number='....1245')

BalanceVar = BalanceVar - 50

(Update Account Write(A)
Set Balance = BalanceVar)
WHERE Number='....1245'

(SELECT Balance Read(B)
FROM Account
WHERE Number='....5679')

BalanceVar = BalanceVar+50

(Update Account Write(B)
Set Balance = BalanceVar
WHERE Number='....5679')

UI

Customer | Transfer

Current Account: 1245
500.00

Savings Account: 5679
1000.

Transfer Screen

FROM
=:... .1245

TO
.... ..5679

Amount
...50.00

O.K.

DBMS: The Concept of *Transactions*

"Transfer"

T_1 :

Read (A)

A := A - 50

Write (A)

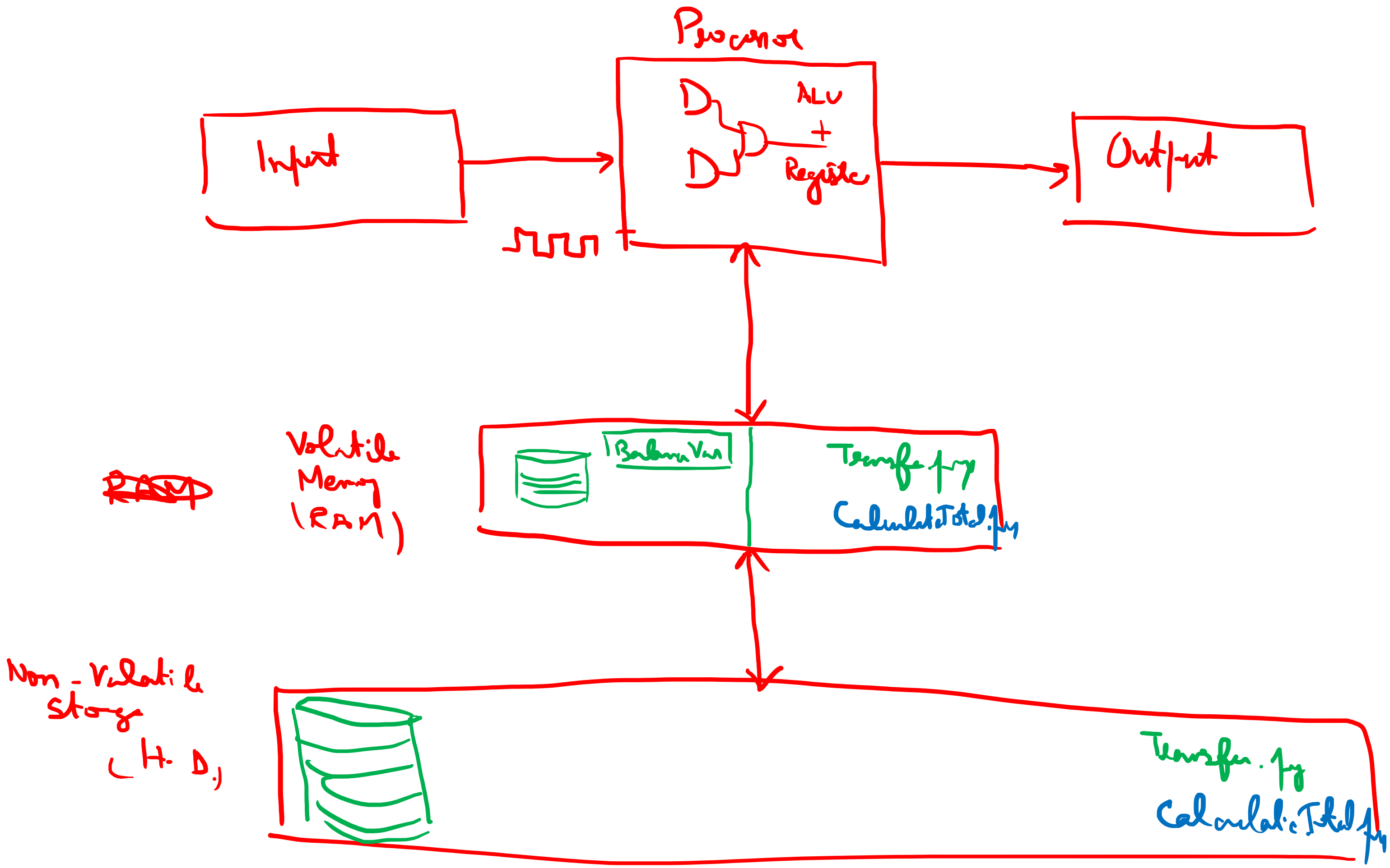
Read (B)

B := B + 50

Write (B)

DBMS: The Concept of *Transactions*

Draw a "Computer" :-



DBMS: The Concept of *Transactions*

- Transaction
  - A transaction is unit of program execution that consists of multiple database operations but appears as a single, indivisible unit from the point of view of the database user/application.
  - A transaction executes in its entirety or not at all.

- Example
  - A transaction to transfer Rs. 50 from account A to account B

T_i: read(A);
$A := A - 50;$
write(A);
read(B);
$B := B + 50;$
write(B).

Transaction Properties: Atomicity

- Transaction to transfer Rs. 50 from account A to account B

$T_{transfer}$

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

- What if the system crashes after step 3 due to a software or hardware failure?

Transaction Properties: Atomicity

- Transaction to transfer Rs. 50 from account A to account B

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

- What if the system crashes after step 3 due to a software or hardware failure?

- "Atomicity" Property/Requirement
  - Either all operations of the transaction are properly reflected in the database or none are.

Transaction Properties: Atomicity

- Transaction to transfer Rs. 50 from account A to account B

  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B := B + 50$
  6. **write**(B)

- What if the system crashes after step 3 due to a software or hardware failure?

- Atomicity Property/Requirement
  - Either all operations of the transaction are properly reflected in the database or none are.
- How?
  - Logs: database system keeps track (on disk) of the old values of any data on which a transaction performs a write.
  - After recovery from the system crash, the database system restores the old values from the log to make it appear as though the transaction never executed.

Transaction Properties: Durability

- Transaction to transfer Rs. 50 from account A to account B

1.	**read**(A)
2.	$A := A - 50$
3.	**write**(A)
4.	**read**(B)
5.	$B := B + 50$
6.	**write**(B)

- Once the user has been notified that the transaction has completed (i.e., the transfer of the Rs.50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

- Durability Property/Requirement
  - After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction Properties: Durability

- Transaction to transfer Rs. 50 from account A to account B

  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B := B + 50$
  6. **write**(B)

- Once the user has been notified that the transaction has completed (i.e., the transfer of the Rs.50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

- Durability Property/Requirement
  - After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

- How?
  - The updates carried out by the transaction should have been written to disk before the transaction completes

Transaction Properties: Isolation

- Transaction to transfer Rs. 50 from account A to account B

transfer.py

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B)*

- What if there is another transaction: Calculate total balance

CalculateTotal.py

1. **read**(*A*)
2. **read**(*B*)
3. print (A+B)

DB_A = 100 DB_B = 100

read (A)
var_A1 = 100
A := A - 50
var_A1 = 50
read (A)
var_A2 = 100
read (B)
var_B2 = 100
print (A+B)
Prints 200
write (A)
DB_A = 50
read (B)
var_B1 = 100
B := B + 50
var_B1 = 150
write (B)
DB_A = 150

DB_A = 100 DB_B = 100

var_A1 = 100
read (A)
var_A1 = 50
A := A - 50
DB_A = 50
write (A)
var_A2 = 50
read (A)
var_B2 = 100
read (B)
print (A + B)
Prints 150
read (B)
var_B1 = 100
B := B + 50
var_B1 = 150
write (B)
DB_B = 150

Transaction Properties: Isolation

- Transaction to transfer Rs. 50 from account A to account B

1. **read**(*A*)
2. $A := A - 50$
3. **write**(*A*)
4. **read**(*B*)
5. $B := B + 50$
6. **write**(*B*)

- What if there is another transaction: Calculate total balance

1. **read**(*A*)
2. **read**(*B*)
3. print (A+B)

- Isolation Property/Requirement
  - For every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
  - Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

Transaction Properties: Consistency

- Transaction to transfer Rs. 50 from account A to account B

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

Transaction Properties: Consistency

- Transaction to transfer Rs. 50 from account A to account B

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B :=$ $\underline{B + 40}$
6. **write**(B)

- What happens if there is logical mistake by the programmer in the coding of Transaction?

Transaction Properties: Consistency

- Transaction to transfer Rs. 50 from account A to account B

  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B + 40*
  6. **write**(*B*)

- What happens if there is logical mistake by the programmer in the coding of Transaction?
  – This is a violation of *consistency requirements*.
  – Examples of explicit consistency requirements: primary keys and foreign keys
  – Examples of implicit consistency requirements : sum of balances of all accounts must be preserved

Transaction Properties: Consistency

- Transaction to transfer Rs. 50 from account A to account B

  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B :=$ <u>$B + 40$</u>
  6. **write**(B)

- What happens if there is logical mistake by the programmer in the coding of Transaction?
  - This is a violation of <u>*consistency requirements*</u>.

- Consistency Property/Requirement
  - A transaction should be "consistency preserving," meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another

Transaction Properties: Consistency

- Transaction to transfer Rs. 50 from account A to account B

1.	**read**(A)
2.	$A := A - 50$
3.	**write**(A)
4.	**read**(B)
5.	$B :=$ <u>$B + 40$</u>
6.	**write**(B)

- What happens if there is logical mistake by the programmer in the coding of Transaction?
  - This is a violation of <u>*consistency requirements*</u>.

- Consistency Property/Requirement
  - A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another
  - Ensuring consistency of an individual transaction is the responsibility of the application programmer who codes the transaction. This task may be facilitated by automated testing of integrity constraints (e.g. primary key and foreign key constraints) by the DBMS.

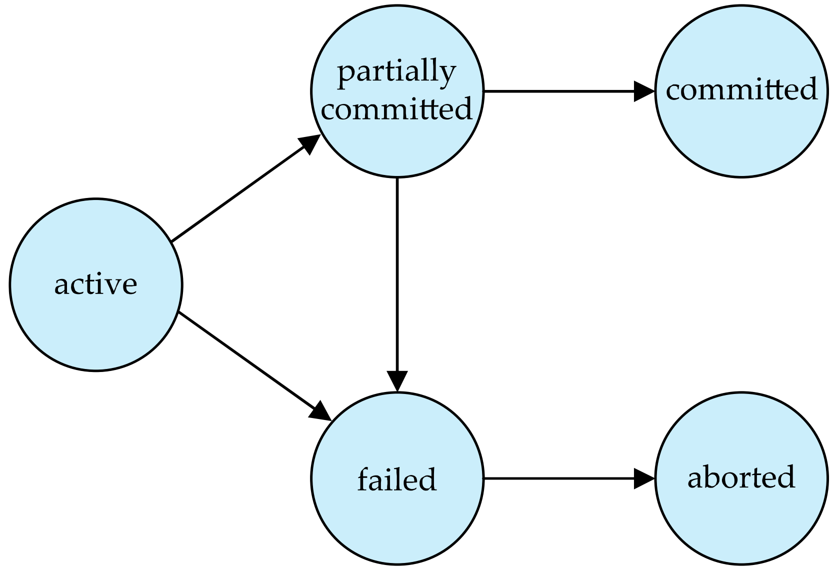
Transaction Properties: ACID

- A transaction must have the following four properties:
  - 1) **A**tomicity,
  - 2) **C**onsistency,
  - 3) **I**solation,
  - 4) **D**urability.

- These form the acronym **ACID** properties.

Transaction States

- Active
  - the initial state; the transaction stays in this state while it is executing.
- Partially committed
  - after the final statement has been executed.
- Failed
  - after the discovery that normal execution can no longer proceed.
- Aborted
  - after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- Committed
  - after successful completion.



TRANSFER:-

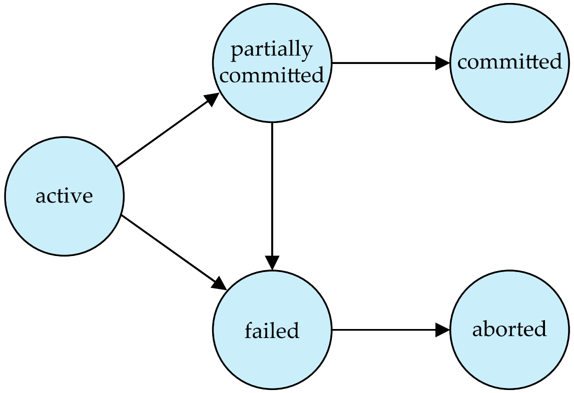
1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

send.email(   )

Transaction States and "Observable External Writes"

- Observable External Writes
  - Once such a write has occurred, it cannot be undone since it may have been seen external to the database system
  - Examples: Writes to a user screen, sending email.

- Most systems allow such writes to take place only after the transaction has entered the committed state.



DIVE DEEP INTO "ISOLATION" Property

Serial vs Concurrent Execution of Transactions

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

T_1
read(A)
$A := A - 50$
write(A)
read(B)
$B := B + 50$
write(B)
commit

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

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)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

Serial vs Concurrent Execution of Transactions

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

T_1
read(A)
$A := A - 50$
write(A)
read(B)
$B := B + 50$
write(B)
commit

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

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)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

- Restricting ourselves to executing transactions serially (i.e. one after the other) makes it easy to achieve isolation among transactions.
- However, concurrent execution of transactions provides significant performance benefits:
  - Increased throughput
  - Reduced average response times

Advantage of Concurrent Execution of Transactions

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

T_1
read(A)
$A := A - 50$
write(A)
read(B)
$B := B + 50$
write(B)
commit

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

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)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

1) • Increased throughput (number of transactions per unit time)
  – one transaction can be using the CPU while another is reading from or writing to the disk

2) • Reduced average response time
  – short transactions need not wait behind long ones.

Concurrent Execution of Transactions: Role of Concurrency-Control Schemes

- "Concurrency-control schemes"
  - Mechanisms to achieve isolation among concurrently-executing transactions
  - Mechanisms to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
- Will study these schemes after studying the notion of ‘correctness of concurrent executions’

T_1
read(A)
$A := A - 50$
write(A)
read(B)
$B := B + 50$
write(B)
commit

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

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)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit