DATABASE SYSTEMS

Database Concurrency

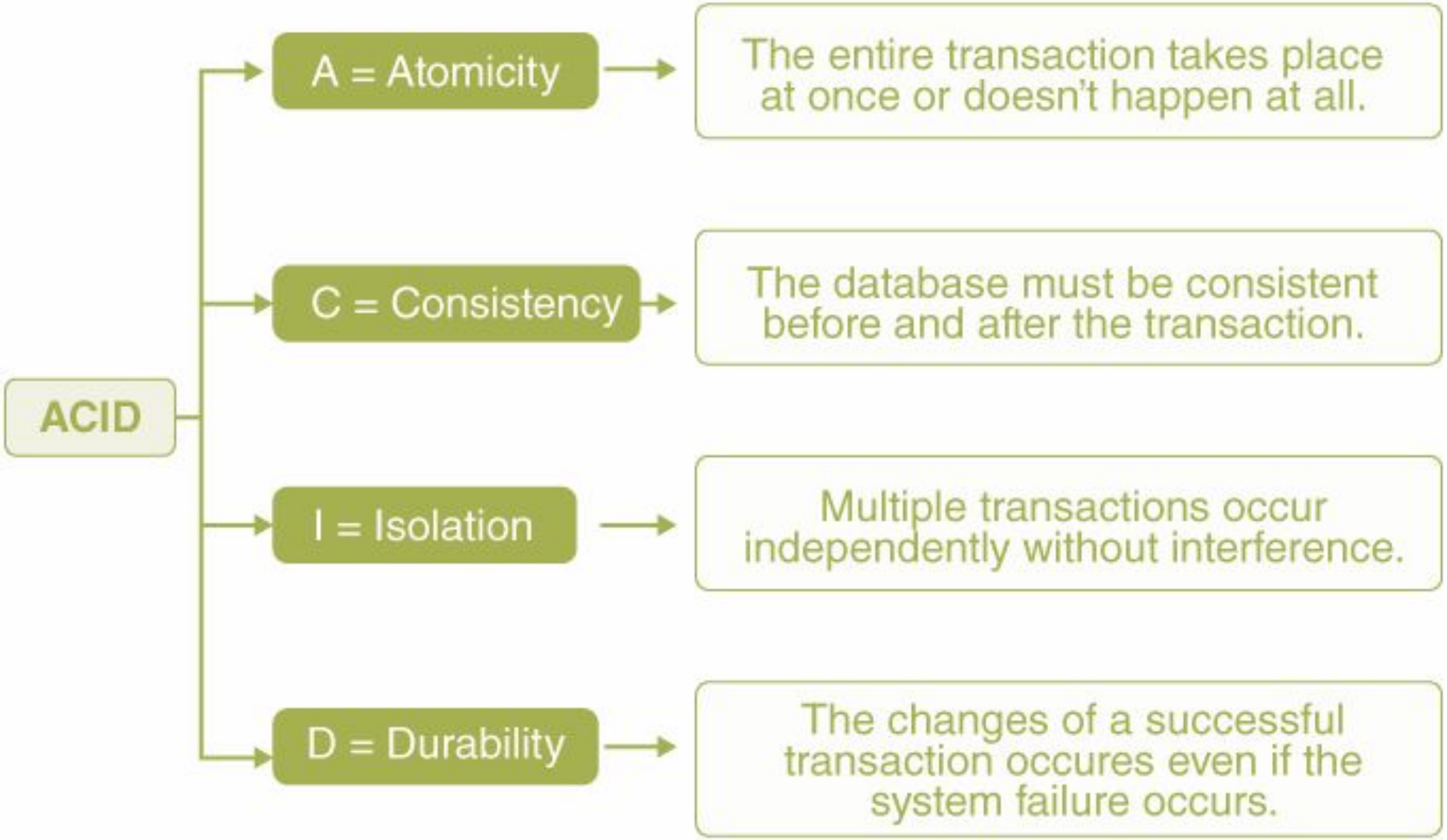
Faculty of AI & MMG

Concurrency Control in Databases

- The **transaction** refers to a small unit of any given program that consists of various low-level tasks.
- Every transaction in DBMS must maintain ACID – A (Atomicity), C (Consistency), I (Isolation), D (Durability).
- One must maintain ACID so as to ensure completeness, accuracy, and integrity of data.

- **Concurrency control** ensures that multiple transactions can execute simultaneously without leading to data inconsistency.
- It maintains the correctness of data when multiple users access/modify it concurrently.

ACID Properties in DBMS



ACID Properties

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** 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.
  - That is, 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.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Example of Fund Transfer

- Transaction to transfer \$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*)

- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database

- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	T2
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A), read(B), print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

Key Concepts in Concurrency Control:

- Locking: This involves using locks to control access to data items, ensuring that only one transaction can modify a specific resource at a time.
- Timestamp Ordering: This method assigns a timestamp to each transaction and orders them based on these timestamps to resolve conflicts.
- Deadlock Prevention: This involves detecting and resolving deadlocks, which can occur when two or more transactions are blocked indefinitely, waiting for each other to release resources.

Deadlocks & Serializability

- **Deadlocks**: Occur when transactions wait indefinitely for each other's locks.
- *Example*: T1 holds A and waits for B; T2 holds B and waits for A.
  - **Solution**: Timeouts or deadlock detection (aborting one transaction).
- **Serializability:** Ensures concurrent transactions produce the same result as some serial execution.
  - Conflict Serializability: Reorder transactions using swap of non-conflicting operations.
  - View Serializability: Less strict, allows blind writes.