DATABASE SYSTEMS

CS – 355/CE – 373

Instructor: Maria N. Samad

October 21st, 2024

TRANSACTION

- A collection of operations that form a ***single***, ***logical unit of work***
- It appears as one task for the user
- For example, transferring funds from a checking account to a savings account
- This is one task for the customer, but in order to perform this task, a database system requires multiple operations, like removing values from the checking account, adding values to the savings account, notifying the customer, etc.
- It is essential that when these operations occur, they should occur holistically, and not in parts.
- Thus, in the funds transfer example mentioned above, it shouldn’t be the case that it removes the funds from the checking account, and stops the transaction without adding to the savings account
- This will lead to inconsistencies and flawed systems

TRANSACTION

- Therefore, if for any reason a system fails after debiting the checking account and before the money was credited in the savings account, the money should be credited back to the checking account to avoid loss
- That is, the system should restore to its previous correct state
- In short, a database system **MUST** ensure proper execution of transactions despite failures – either a transaction is executed completely, or none at all
- The system should also manage concurrent execution of transactions in a way that prevents inconsistencies in data, even when multiple users are accessing the system concurrently, i.e. at the same time

INTRODUCTION

- Transaction processing systems are systems with large databases and hundreds of concurrent users executing database transactions.
- Examples of such systems include
  - Airline reservations
  - Banking
  - Online retail purchasing
  - Stock markets
  - Supermarket checkouts, etc.
- These systems require high availability and fast response time for hundreds of concurrent users.

MOTIVATION

- Several problems can arise if concurrent transactions occur in an uncontrolled manner.
- For example, consider an airline reservation system, where two users concurrently update the database.
- It is quite possible that because of concurrent access, the updated values might be incorrect.
- We present some examples in the subsequent slides.

EXAMPLE

- If the two transactions accessing the same data item in database occurs on separate time, it won't be an issue.
- For example, assume a transaction T1 that transfers N reservations from one flight whose number of reserved seats is stored in the database item named X to another flight whose number of reserved seats is stored in the database item named Y.
- Another transaction T2 that just reserves M seats on the first flight (X) referenced in transaction T1.
- To simplify our example, we do not show additional portions of the transactions, such as checking whether a flight has enough seats available before reserving additional seats.

EXAMPLE

- The two transactions T1 and T2 can be defined as follows:

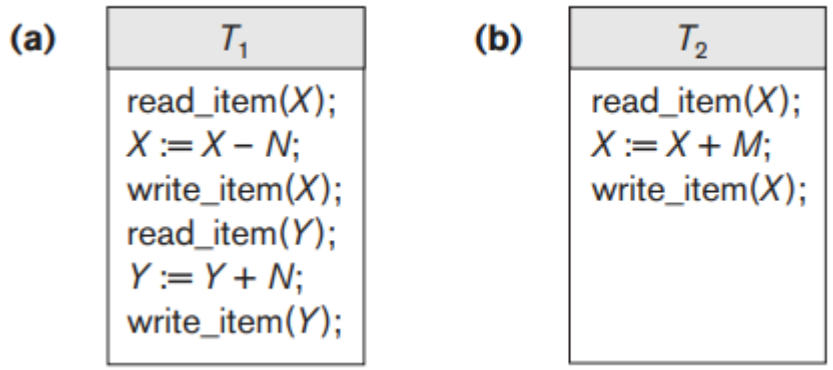


Figure 20.2
Two sample transactions.
(a) Transaction T_1.
(b) Transaction T_2.

- Assume X = 80, Y = 50, N = 5 and M = 4, initially
- Also assume that T1 happens before T2
- Output?

INTERLEAVED CONCURRENCY

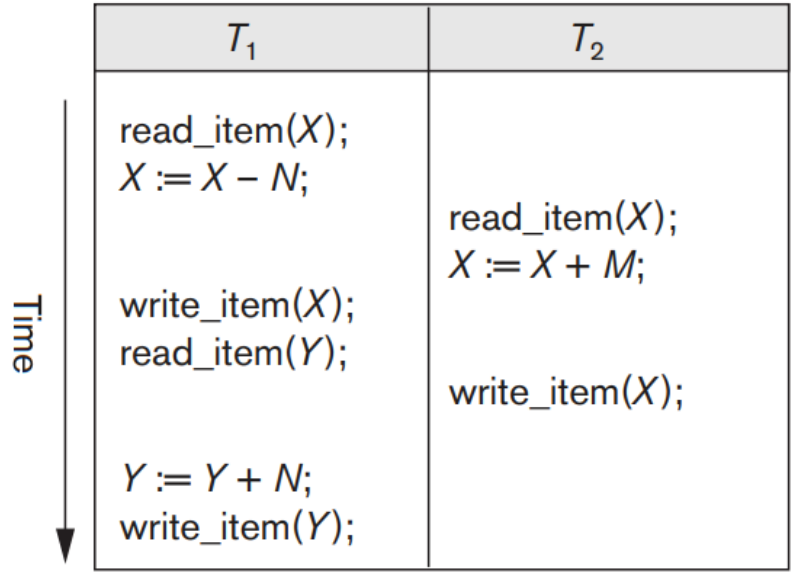
- But it’s not necessary that the transactions will **not** occur simultaneously
- Concurrent systems perform multiple transactions at a time
  - Using multiprogramming operation system concepts
- The transactions are ***interleaved*** such that some commands from one process are executed, then that process is suspended, and then it executes some commands from the next process, and so on
- This process is called ***interleaved concurrency***

INTERLEAVED CONCURRENCY

- However, this can lead to different problems and inconsistencies of data
- The three major problems are:
  - Lost Update Problem
  - Temporary Update Problem
  - Incorrect Summary Problem

THE LOST UPDATE PROBLEM

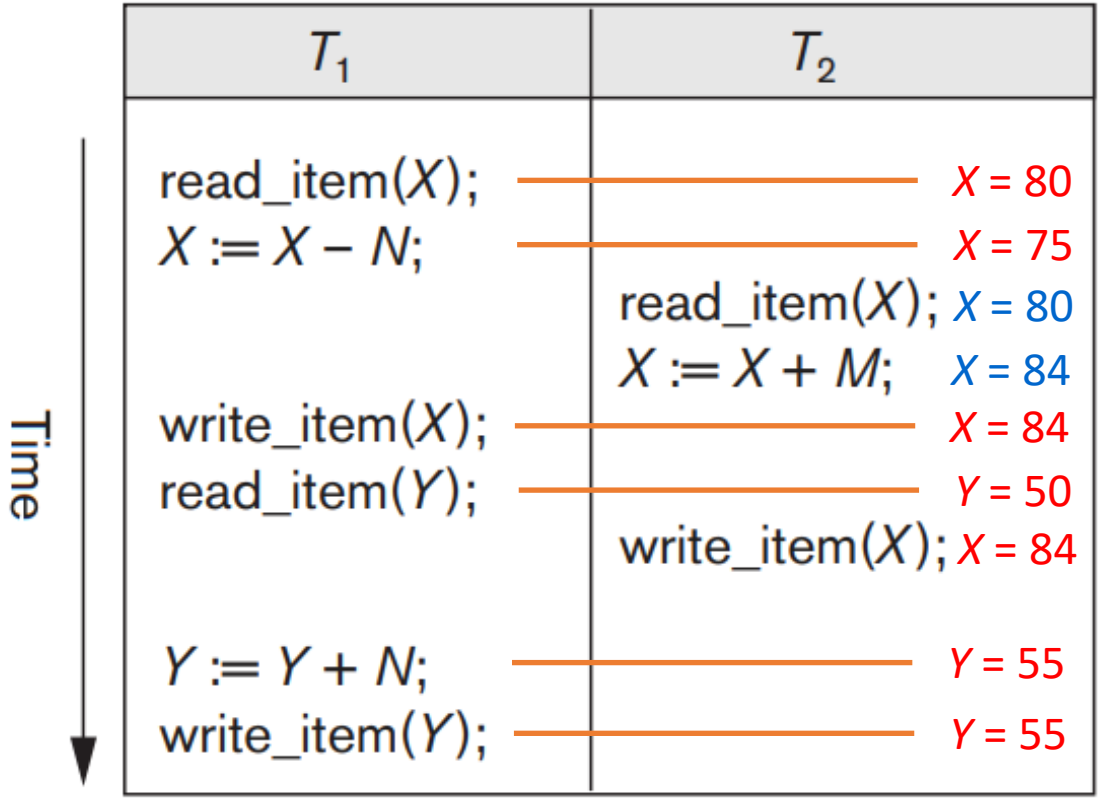
- This problem occurs when two transactions that accesses the same database item, and their operations are interleaved in such a way that makes the value of some of these items incorrect
- Using the same example of airline reservation system from before, but now happening concurrently by interleaving the different operations



THE LOST UPDATE PROBLEM

Consider an airline reservation system where users concurrently update the database.

- Let $X = 80$, $Y = 50$, $N = 5$, $M = 4$.
- T_1 removes N seats from X and adds N seats to Y, while T_2 adds M seats to X.
- At the end of the operations, the correct values should be
  - $X = 80 – 5 + 4 = 79$,
  - $Y = 50 + 5 = 55$.



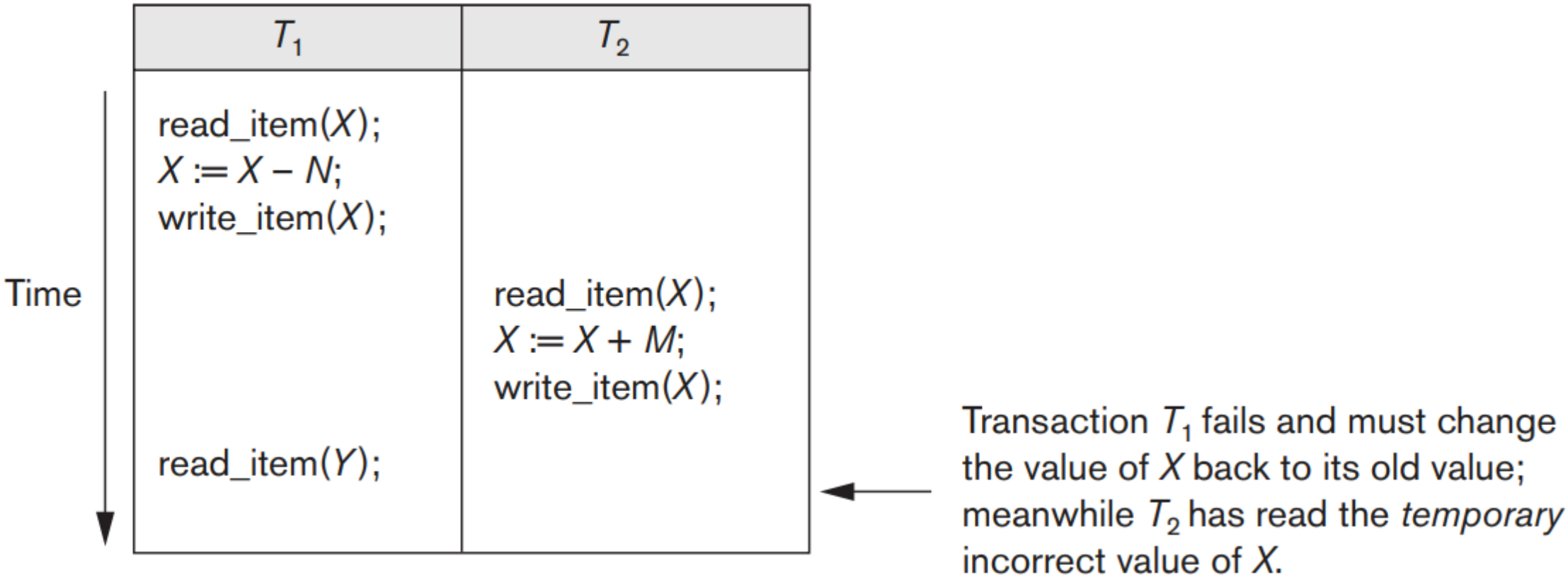
THE LOST UPDATE PROBLEM

- In the previous example, the correct value for *X* should have been 79 (5 seats removed, 4 seats added).
- However, the value written to the database was *X* = 84.
- The final value of *X* is incorrect because T_2 reads the value of *X* before T_1 changes it in the database,
- Therefore, the updated value resulting from T_1 is lost.
- This is known as the lost update problem.

THE TEMPORARY UPDATE PROBLEM

- This problem occurs when one transaction updates the database but before the transaction is completed, it fails for some reason
- Meanwhile the other transaction accesses this updated item before it was reverted back to its original state
- Commonly known as the ***dirty read problem***

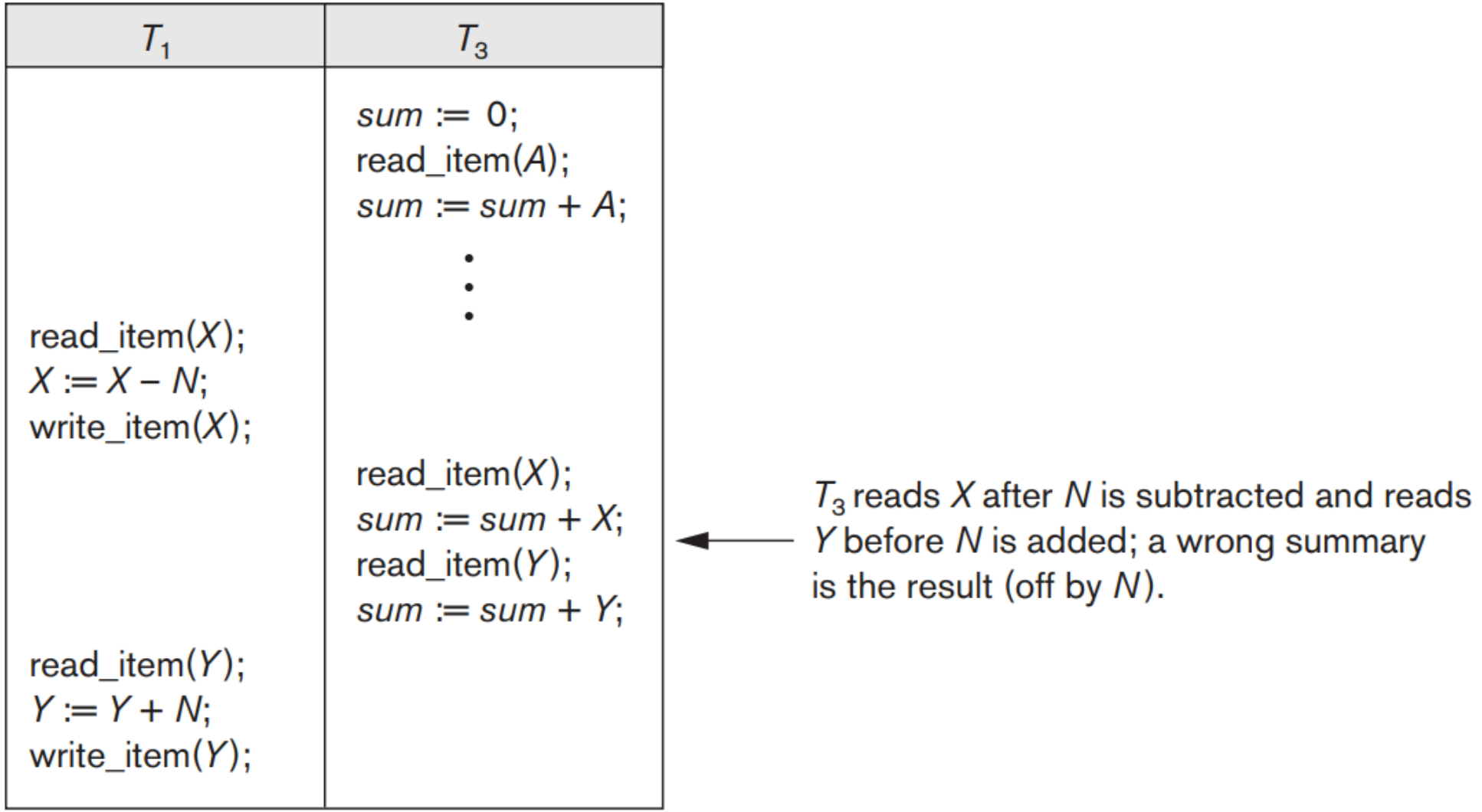
THE TEMPORARY UPDATE PROBLEM



INCORRECT SUMMARY PROBLEM

- This problem occurs when one transaction is calculating an aggregate value of the data items in the database, while another is updating one or all of these data items
- You end up with an incorrect answer even when the database is correctly updated

INCORRECT SUMMARY PROBLEM



TRANSACTIONS

- A ***transaction*** is a unit of program execution that accesses and possibly updates various data items.
- It is an executing program that forms a logical unit of database processing and includes one or more database access operations — insertion, deletion, modification (update), or retrieval operations.
- A transaction is delimited by statements (or function calls) of the form:
  - **`begin transaction`**
  - **`end transaction`**.

TRANSACTIONS

- A transaction (with its steps) must appear to the user as a single, indivisible unit.
- Since a transaction is indivisible, it either executes in its entirety or not at all.
- Thus, if a transaction begins to execute but fails for any reason, changes to the database made by the transaction must be undone.
- This requirement holds regardless of whether the transaction encountered an error, or there is a hardware crash.

ACID PROPERTIES

- A transaction must have the following four properties:
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

- These form the acronym **ACID** properties and are discussed in the subsequent slides.

ACID PROPERTIES – ATOMICITY

Atomicity

- Either all operations of the transaction are reflected properly in the database, or none are.
- A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all .
- Ensuring atomicity is difficult since some changes to the database may still be stored only in the main-memory variables of the transaction, while others may have been written to the database.
- This “all-or-none” property is referred to as atomicity.

ACID PROPERTIES – CONSISTENCY

Consistency

- Execution of a transaction in isolation (i.e., with no other transaction executing concurrently) preserves the consistency of the database.
- 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.
- An example of a consistency preserving transaction is: if a transaction transfers an amount of *N* PKR from Bank Account *A* to *B*, then the sum of the amounts in the Bank Accounts *A* and *B* must be the same before and after the transaction.

ACID PROPERTIES – ISOLATION

Isolation

- Each transaction is unaware of other transactions executing concurrently in the system.
- The execution of a transaction should not be interfered with by any other transactions executing concurrently.
- The isolation property ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that would have been obtained, had these transactions executed one at a time, i.e. in isolation.

ACID PROPERTIES – DURABILITY

Durability

- After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
- The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.
- The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

STORAGE STRUCTURE

To ensure the ACID properties of a transaction, we classify storage as follows:

- **Volatile** storage: Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory.
- **Non-volatile** storage: Information residing in non-volatile storage survives system crashes. Examples of non-volatile storage include secondary storage devices.
- **Stable** storage: Information residing in stable storage is never lost. Although theoretically impossible to obtain, it can be approximated by techniques that make data loss extremely unlikely such as replicating information in several non-volatile storage media with independent failure modes.

For a transaction to be durable, its changes need to be written to stable storage.

TRANSACTION STATES

- A transaction that does not complete its execution successfully is termed **aborted**. An aborted transaction must have no effect on the state of the database.
- Any changes that the aborted transaction made to the database must be undone.
- Once the changes caused by an aborted transaction have been undone, the transaction has been **rolled back**.
- A transaction that completes its execution successfully is said to be **committed**. A committed transaction transforms the database into a new consistent state.
- Once a transaction has committed, we cannot undo its effects by aborting it. To undo the effects of a committed transaction we execute a **compensating** transaction.

TRANSACTION STATES

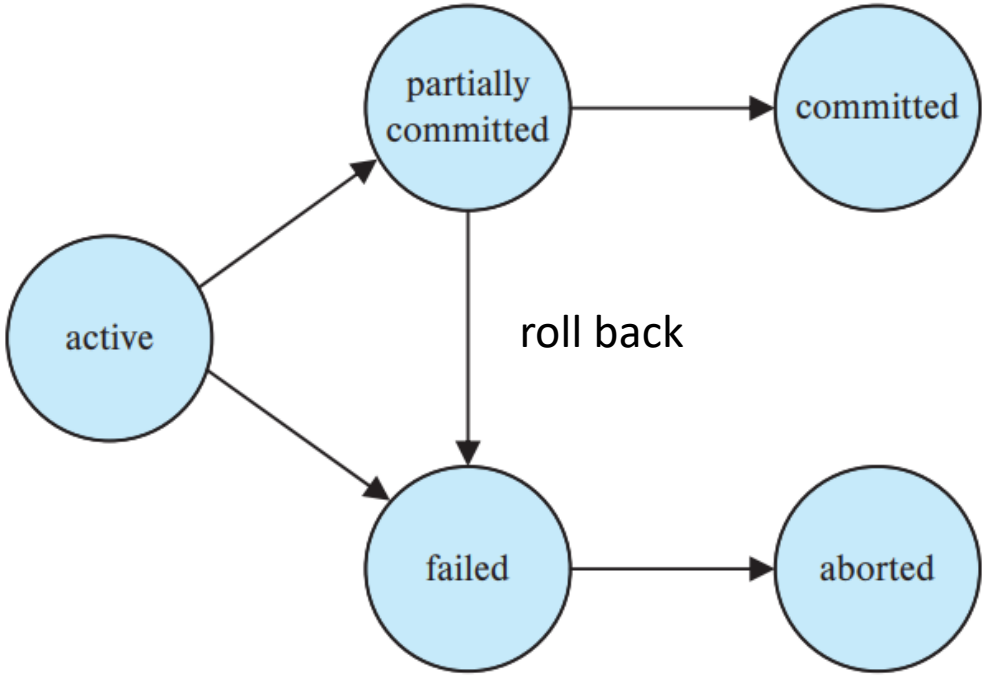


Figure 17.1 State diagram of a transaction.

TRANSACTION STATES

To elaborate upon the completion of a transaction, the following states for a transaction are defined:

- ***Active*** – the initial state; the transaction stays in this state while it is executing.
- ***Partially committed*** – while the final statement is being 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.

TRANSACTION PROCESSING SYSTEMS

- Activity Sheet

TRANSACTION PROCESSING SYSTEMS

- Activity Sheet Solution:
  - Transaction Processing Systems Solution