TRANSACTIONAL PROCESSING

INTRODUCTION

Single-User vs. Multi User Database Systems

1. **SingleSystems-User** :
    - These allow only **one user** to access the database at a time.
    - Typically used on **personal computers**.
    - **Example:** A small program where only one person needs to work with the database at any given moment.
2. **Multiuser Systems**:
    - Allow **multiple users** to access the database **simultaneously**.
    - Common in **large organizations like banks, airlines, supermarkets, and insurance agencies**, where many users need to work with the same data at the same time.
    - **Example:** An airline reservation system where hundreds of agents book tickets at the same time.

How Multi User Access Works

- **Key Concept: Multiprogramming**
    - A computer system can allow multiple users to work at the same time because of **multiprogramming**.
    - Multiprogramming enables the **computer's operating system** to manage **several tasks** (or "processes") at once by **switching** between them.

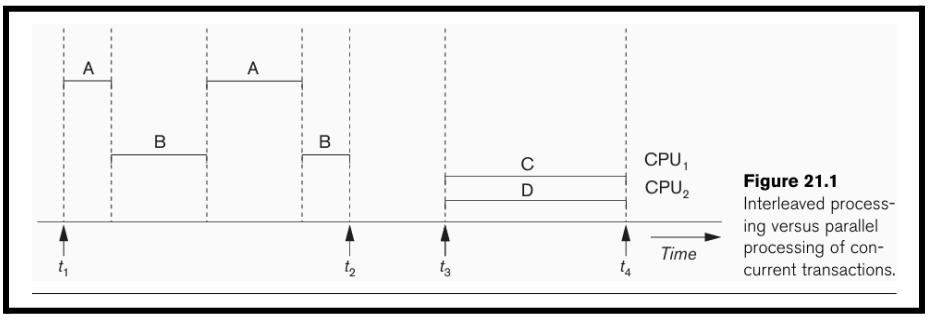


Figure 21.1 Interleaved processing versus parallel processing of concurrent transactions.

    - If one CPU is present- **INTERLEAVING.**
    - If more than 1 CPU is present- **PARALLEL PROCESSING.**
- A computer's **CPU (Central Processing Unit)** can only work on **one task at a time**.
- However, with **Interleaving**, the CPU switches between different tasks quickly, giving the illusion that tasks are running at the same time.
    - **Example of Interleaving**:
        - Task A runs for a few moments, then pauses.
        - Task B runs, then pauses.
        - The CPU keeps switching between A and B, resuming where each task left off.

- **When Multiple CPUs Are Present**:
    - If the system has more than one CPU, multiple tasks can truly run in **parallel** (at the same time).
    - Example: Tasks C and D running simultaneously on two separate CPUs.

Concurrency in Multi user Databases

- **Concurrency** means **multiple tasks or processes happening at the same time** within a system. However, it doesn't necessarily mean these tasks are running simultaneously—it often refers to tasks being managed in a way that they appear to run at the same time.
- This concurrency is **carefully controlled** to ensure:
    1. **No data conflicts** occur when multiple users try to **update the same information.**
    2. All users see **consistent and accurate data.**

WHAT IS A TRANSACTION?

- A **transaction** is a **program** or a **set of database operations** that forms a **logical unit of work**.
- Transactions perform tasks such as **insertion**, **deletion**, **modification/update**, or **retrieval/access** of data.
- Transactions ensure that **database operations are executed completely and accurately**.

Types of Transactions

1. **Read-Only Transaction**:
    - Retrieves/accesses data without making any changes.
    - Example: **Checking your bank account balance.**
2. **Read-Write Transaction**:
    - Both reads and modifies the database.
    - Example: **Transferring money between bank accounts.**

How Transactions Work

- **Defining Transaction Boundaries**:
    - Transactions begin and end with specific commands like **BEGIN TRANSACTION** and **END TRANSACTION**.
    - All **operation**s performed **between these commands** are treated as **one transaction.**
    - A **single program** can have **multiple transactions** if it **defines multiple boundaries.**
- Two basic database operations are commonly used in transactions:
    1. **read_item(X):** Reads the value of a data item (e.g., record, field) named X.
    2. **write_item(X):**Updates the value of a data item named X.

1. **For read_item(X):**
    - Locate the disk block containing X.

    - Load the disk block into the buffer of main memory (if not already there).
    - Copy the value of X into a program.
2. **For write_item(X):**
    - Locate the disk block containing X.
    - Load the disk block into the buffer of main memory (if not already there).
    - Modify X in the **memory buffer**.
    - Write the updated data back to the disk.

Note: Writing changes back to disk may not happen immediately. It depends on the **recovery manager** and **buffer replacement policies**.

DBMS Buffers

- A **DBMS buffer** is a part of the **main memory** used to **temporarily hold disk blocks.**
- Buffers are important for improving performance:
    - Database operations (read or write) work directly on these in-memory buffers.
    - If buffers are **full**, a **buffer replacement policy** determines which buffer to replace or the **recovery manager** decides.
- **Buffer Operations**: (same as above).

Database item/ Data item

- A **database item** is the smallest unit of data that a **transaction can access.**
- A **record**, **field**, or **disk block** are used to represent a data item.
- Each data item has a **unique name** (e.g., disk block address) fo**r identification.**

Granularity of Data Items

- **Granularity** refers to the **size of a data item**.
- A **disk block** (large unit).
- A **record** (medium unit).
- A field of a record (small unit).

Concurrency in Transactions

- **Concurrency**: Transactions often run at the same time (concurrently) in a multiuser environment.
- **Problem**: If concurrency is not properly controlled, multiple transactions may conflict, leading to:
    - **Inconsistent data, inaccurate data**. (due to overwriting of data)
    - **Errors or failures in database operations.**
    - **Data conflicts**/ Transaction Conflicts.

Example of Transactions

Consider two simple transactions:

1. **Transaction T1**:
    - Reads items X and Y.
    - Updates X and Y.
2. **Read-Set**: {X, Y} (items read by T1)
   Write-Set: {X, Y} (items written by T1)
3. **Transaction T2**:
    - Reads and modifies another set of data items.

- If T1, T2 access the **same data** (e.g., X or Y) at the **same time,** without proper control:
    - They could **overwrite** each other's changes.
    - This might result in an **inconsistent database**.

Concurrency Control Mechanisms ensure:

1. Transactions don't interfere with each other.
2. The database remains **accurate** and **consistent**.

WHY CONCURRENCY CONTROL IS NEEDED

When multiple transactions are executed simultaneously in a database (concurrently), issues can arise if there is no proper **concurrency control**. Below are the main problems that can occur:

1. The Lost Update Problem

- **What happens?**
    - Two transactions access the **same data item**.
    - Their operations **overlap/ interleave** in a way that **one of the transaction's updates gets overwritten or lost**.
- **Example**:
    - Suppose there are 80 reserved seats on flight X. (X = 80).
    - **Transaction T1** transfers 5 seats from flight X to another flight Y (N = 5).
    - **Transaction T2** reserves 4 new seats on flight X (M = 4).
    - If T2 reads the value of X **before** T1 completes its update, T2's operation overwrites the value updated by T1.
    - **Expected Result**: $X=80-5+4=79$.
    - **Actual Result** (due to interleaving): X=84. (T1's update is lost).

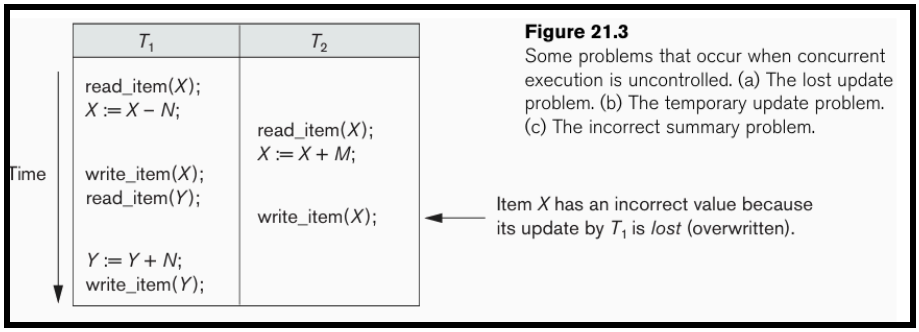
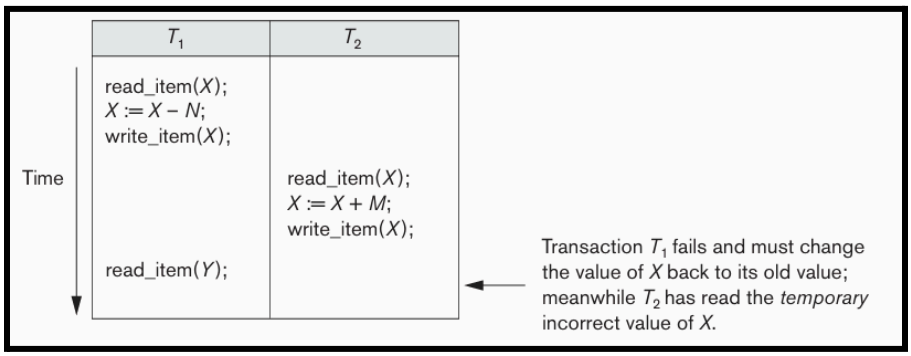


Figure 21.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

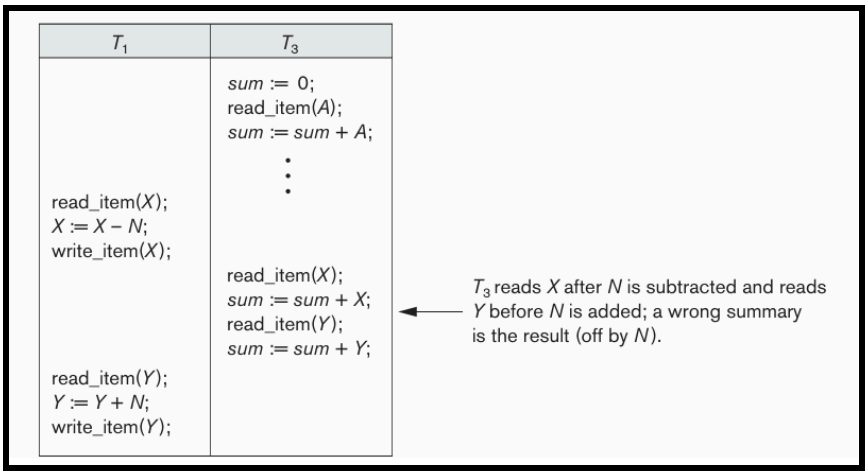
2. The Temporary Update (Dirty Read) Problem

- **What happens?**
  - A transaction **updates a data item** but **fails** for some reason **before completing**.
  - Meanwhile, another transaction reads this **temporary (dirty)** value, which will later be discarded.
- **Example**:
  - **Transaction T1** updates the value of X.
  - T1 fails before completion and must change the value to the old value of X.
  - T2 ends up reading the incorrect/Temporary value before T1 changes its value.
  - T2 is now working with a value that is invalid (dirty data).



3. The Incorrect Summary Problem

- **What happens?**
  - A transaction i**s calculating a summary or aggregate function** while **another transaction** is **updating the same data item**s, leading to **partial or inconsistent results**.
  - The **aggregate function may calculate** some values **before they are updated** and others **after they are updated.**
- **Example**:
  - **Transaction T3** is calculating the total number of reservations across all flights.
  - Meanwhile, **Transaction T1** transfers N reservations from flight X to flight Y.
  - If T3 reads the value of X **after** seats are subtracted but reads Y **before** seats are added, the total is off by N.



4. The Unrepeatable Read Problem

- **What happens?**
  - A transaction **reads the same data item twice**, but **another transaction updates the item between the two reads**, causing it to return **different values.**
- **Example**:
  - Transaction T1: A customer initiates **Transaction T1** to check seat availability on a flight. **T1 reads the value of X** (e.g., `X = 50`, representing 50 available seats on the flight).
  - Another Transaction Updates X Before T1 completes, modifies the number of seats on the same flight, updating X (e.g., `X = 48` because 2 seats are reserved).
  - Later in **T1**, the customer decides to proceed with the reservation and reads X again.
  - This time, **T1 sees a different value of X** (e.g., `X = 48`), which does not match the earlier value of `50`.

WHY RECOVERY IS NEEDED?

When a transaction is executed, the **DBMS** ensures that either:

1. **All operations** of the transaction are completed successfully and their results are permanently recorded in the database (**committed**), or
2. **None** of the operations have any effect on the database (**aborted**).

This is essential because a transaction is a **logical unit of work**, and partial execution can lead to inconsistency. If a **transaction fails before completion**, the **changes it made to the database** must be **undone to maintain consistency.**

Types of Failures

1. **Computer Failure (System Crash)**:
    - **Hardware, software, or network errors** may interrupt a transaction.
    - Example: **Main memory failure**, **operating system crash.**
2. **System Errors**:
    - Errors such as **division by zero**, **integer overflow,** or **invalid input** may cause a transaction to fail.
    - **Logical programming errors** or **user intervention** (e.g., canceling a transaction) can also abort the transaction.
3. **Local Errors or Exception Conditions**:
    - During execution, certain conditions might prevent the transaction from continuing:
        - **Missing data** for the **operation.**
        - Exception conditions like **insufficient account balance during a withdrawal.**
    - These are often programmed in the transaction itself.
4. **Concurrency Control Enforcement**:
    - To maintain **serializability**, the DBMS may **abort transactions** that **violate concurrency rules** or to **resolve deadlocks** (when transactions block each other).
    - Such transactions are usually **restarted automatically.**
5. **Disk Failures**:
    - **Disk blocks** may **lose data** due to **read/write malfunctions** or **disk crashes**.
    - These failures **affect data integrity** during transaction execution.
6. **Physical Problems and Catastrophes**:
    - Issues like **power failure, fire**, **theft**, **sabotage**, or **accidental overwriting of data**.
    - These catastrophic failures are **rare** but can cause **significant damage.**

For failures of types **1 to 4** (more common):

- The **DBMS must maintain enough information to quickly recover** from the failure.
- This ensures the **database remains consistent** and,
- **transactions that fail do not leave partial changes.**

For failures of types **5 and 6** (less common but severe):

- Recovery requires more extensive measures because data integrity may be compromised.

TRANSACTION STATES AND ADDITIONAL OPERATIONS

A transaction is a **logical unit of work** in a database system. It must be executed **completely** (committed) or **not at all** (aborted). The **system keeps track of the transaction's lifecycle** using s**pecific operations** and **states**.

Operations in Transactions

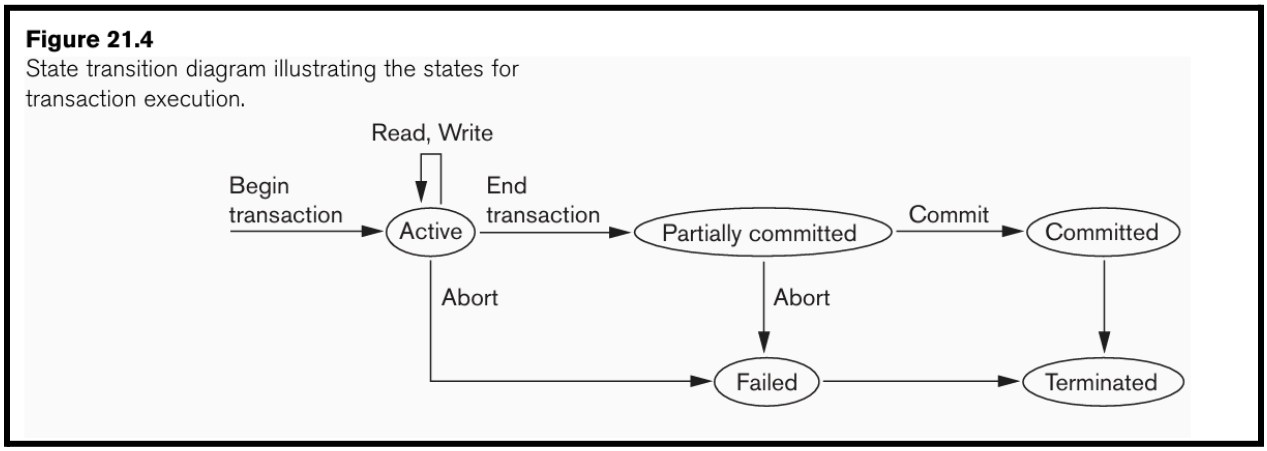
1. **BEGIN_TRANSACTION**:
    - Marks the **start of the transaction**.
    - The system begins **tracking the transaction's execution.**
2. **READ or WRITE**:
    - These operations allow the transaction to:
        - **READ**: Retrieve data from the database.
        - **WRITE**: Update or insert data into the database.
3. **END_TRANSACTION**:
    - Indicates the **end of the transaction's operations (READ/WRITE).**
    - At this point, the system decides whether:
        - The transaction can be permanently saved to the database (**COMMITTED**), or
        - The transaction must be undone (**ABORTED**).
4. **COMMIT_TRANSACTION**:
    - Signals that the transaction has been **successfully completed.**
    - All changes made by the transaction are now **permanent in the database.**
5. **ROLLBACK (or ABORT)**:
    - Signals that the transaction has **failed or been aborted.**
    - All **changes made by the transaction** are **undone**, ensuring the **database** is restored to its previous **consistent state.**

Transaction States

1. **Active State:**
    - The transaction enters this state after **BEGIN_TRANSACTION**.
    - It **performs its operations (READ/WRITE)** while in this state.
2. **Partially Committed State:**
    - After completing its operations (**END_TRANSACTION**), the transaction enters this state.
    - At this point, the **system performs checks (e.g., concurrency or recovery protocols)** to ensure the **transaction's changes can be safely committed.**
3. **Committed State:**
    - If **all checks are successful,** the transaction reaches this state.
    - Changes made by the transaction are now **permanently recorded in the database.**
    - **Even if a system failure** occurs after this point, the **changes will remain intact.**
4. **Failed State:**
    - If the transaction fails due to an error (e.g., **system crash, serializability violation)** or is explicitly aborted, it enters this state.
    - The **transaction's changes must be undone** via **ROLLBACK**.
    - Database goes back to its old state to maintain consistency.
5. **Terminated State:**
    - After the transaction is **either committed or rolled back,** it enters this state.
    - The **system removes any information** about the **transaction** from its **tracking tables.**

- **Failed or aborted transactions** can be **restarted** later as **new transactions.**

Key Transitions in Transaction States



- A transaction moves from **Active → Partially Committed → Committed** if it succeeds.
- If a failure occurs, the transaction moves from **Active → Failed → Terminated** (after rollback).

Why is This Important?

The transaction lifecycle ensures:

1. Database **consistency**.
2. The ability to **recover from failures.**
3. That **no incomplete or inconsistent changes** are left in the **database**.

THE SYSTEM LOG

The **system log** is a **crucial component in a database system** to **ensure recovery from failures.**

It **tracks all operations performed by transactions** that **affect the database values.**

- The log ensures that the system can:
    1. **Recover the database to a consistent state after failures.**
    2. **Undo or redo specific transaction operations based on the log entries.**
- It is maintained as a **sequential, append-only file** stored on disk, unaffected by most failures (except disk or catastrophic failures).

How the Log Works

1. **Log Storage**:
    - **Log entries** are first written to **main memory buffers**.

- When the **buffer is full** or certain conditions occur, it is appended to the **log file on disk**.
- The log file is **periodically backed up to archival storage (e.g., tapes)** to protect against **catastrophic failures and also reduces overhead.**

2. **Log Records**:
   - Each transaction is assigned a **unique transaction ID (T)**, which is **included in the log records.**
   - The following types of log records are written to the log:

1. **[start_transaction, T]**:
   - Indicates that transaction T has started execution.
2. **[write_item, T, X, old_value, new_value]**:
   - Indicates that transaction T has updated the value of a database item ***X:***
     - ***old_value:*** The value of ***X*** before the change.
     - ***new_value:*** The value of ***X*** after the change.
3. **[read_item, T, X]**:
   - Indicates that transaction T has read the value of database item ***X.***
4. **[commit, T]**:
   - Indicates that transaction T has successfully completed, and its changes can be **permanently recorded** in the database.
5. **[abort, T]**:
   - Indicates that transaction T has been aborted.

How the Log is Used for Recovery of failures

1. **Undo Operations**:
   - If a **transaction T fails,** its **changes can be undone** by **tracing backward through the log** and **resetting all updated items to their *old_values***.
2. **Redo Operations**:
   - If a **failure occurs after a transaction has been committed** but **before its changes are fully written to disk**, the transaction's operations are **redone** using the ***new_values*** **recorded in the log**.

COMMIT POINT OF A TRANSACTION

The **commit point** is the moment when a **transaction successfully completes all its operations**, and its effects are ready to be **permanently recorded in the database.**

1. **All Operations Are Successful**:
   - The transaction has **executed all its database operations (e.g., READ, WRITE) without errors.**
2. **Effects Are Recorded in the Log**:
   - All **changes** made by the transaction are written to the **system log** for **recovery purposes**.

3. **Commit Record Added:**
    - The transaction writes a **commit record [commit, T]** to the log to signify that it has reached its commit point.
    - After this, the transaction is considered **committed**, and its **changes are permanent.**

What Happens During a System Failure?

- If a failure occurs, the log is used to determine the status of transactions:
    1. **Transactions Without a Commit Record**:
        - If a transaction has a `[start_transaction, T]` record in the log but no `[commit, T]` record, it is considered incomplete.
        - Such transactions must be **rolled back**, **changes are undone.**
    2. **Transactions With a Commit Record**:
        - If a transaction has a `[commit, T]` record, all its changes are already logged.
        - These changes can be **redone** to ensure consistency.

Force-Writing the Log

- **Before a transaction is committed,** all its **log entries (including WRITE operations)** must be written from the **log buffer** in main memory to the **disk**. This process is called **force-writing** the log.
- **Why is this important?**
    - In case of a **system crash, data in main memory might be lost.**
    - Force-writing ensures that the log entries are **safely stored o**n the disk, **making recovery possible.**

DESIRABLE PROPERTIES OF TRANSACTIONS

Transactions in a database system must satisfy specific properties to ensure reliable and consistent operations. These are known as the **ACID properties**: **Atomicity, Consistency, Isolation, and Durability**.

1. Atomicity

- **Definition**: A transaction is an **all-or-nothing** operation. It must either:
    - Be completed in its entirety, or
    - Have no effect at all on the database.
- **Example**:
    - If a transaction debits an amount from one bank account and credits it to another, both actions must occur together. If one fails, the entire transaction is undone.
- **Responsibility**:

- The **recovery subsystem** of the DBMS **ensures atomicity by undoing the effects of incomplete transactions.**

2. Consistency

- **Definition**: A transaction must take the database from one **consistent state** to another. If the database satisfies all rules and constraints before the transaction, it should continue to do so after the transaction completes.
- **Example**:
  - If a banking transaction deducts $500 from one account, the sum of all accounts in the system should remain consistent (e.g., reflecting a transfer, not data loss).
- **Responsibility**:
  - **Programmers** or the **DBMS integrity module** ensure that transactions are designed to maintain database consistency by enforcing rules like **integrity constraints.**

3. Isolation

- **Definition**: A transaction should appear to be executed in **isolation** from other transactions, even i**f multiple transactions are running concurrently.**
  - **Other transactions must not interfere with its execution.**
- **Example**:
  - If two transactions are accessing the same database item (e.g., seat availability in an airline reservation system), they should not see or affect each other's intermediate results.
- **Responsibility**:
  - The **concurrency control subsystem** of the DBM**S ensures isolation** by **preventing interference between transactions.**
- **Levels of Isolation**:
  - **Level 0**: No dirty reads.
  - **Level 1**: No lost updates.
  - **Level 2**: No lost updates and no dirty reads.
  - **Level 3 (True Isolation)**: Ensures repeatable reads in addition to the above.

4. Durability

- **Definition**: Once a transaction is committed, its changes must **persist** in the database, even in the event of a system failure.
- **Example**:
  - After a money transfer transaction is committed, the updated balances in the database must remain intact even if the system crashes.
- **Responsibility**:
  - The **recovery subsystem** ensures durability by **permanently recording changes** to the database, typically by **using the system log.**

SCHEDULES (HISTORIES) OF TRANSACTIONS

A **schedule** (or history) of transactions is the **ordering of operations** (like reading, writing, committing, or aborting) from **multiple transactions**.

These operations may **overlap (interleaving of transactions)**, but the **order must maintain specific rules to ensure consistency and correctness.**

What is a Schedule?

- A schedule **S** defines the **order of operations** from a **set of transactions T1,T2,...,Tn.**
- The **operations from different transactions may be interleaved, but**:
    1. The operations **within a single transaction** must appear in the **same order** as they were **written in the transaction.**
    2. The order of all operations in the schedule is a **total order**, meaning **every pair of operations has a defined sequence.**

Types of Operations in Schedules

- **Notation**:
    1. **bi:** Begin transaction Ti.
    2. **ri(X):** Transaction Ti reads item X.
    3. **wi(X):** Transaction Ti writes item X.
    4. **ci:** Commit transaction Ti.
    5. **ai**: Abort transaction Tii.

Conflicting Operations

Two operations in a schedule are said to **conflict** if they:

1. Belong to **different transactions**.
2. Access the **same data item X**.
3. At least one of the operations is a **write**.

- **Types of Conflicts**:
    1. **Read-Write Conflict**:
        - Example: **r1(X);w2(X).**
        - If their order is reversed **w2(X);r1(X**), the value read by r1(X) changes.
    2. **Write-Write Conflict**:
        - Example: **w1(X);w2(X).**
        - Reversing the order **w2(X);w1(X)** changes the final value of X.
    3. **Non-Conflicting Operations**:
        - **r1(X);r2(X)** Both are reads, so their order does not matter.
        - **w2(X);w1(Y):** These write to different items, so no conflict exists.

Complete Schedules

1. **Contains all operations** of T1,T2,...,Tn, including **either commit (ci) or abort (ai)** for each transaction.
2. **Order of operations** for each transaction is **preserved in the schedule**.
3. **Conflicting operations**:
    - For any two conflicting operations, one must occur before the other in the schedule.

- Example:
    - **Sa: r1(X);r2(X);w1(X);r1(Y);w2(X);w1(Y);c1;c2.**

Committed Projection of a Schedule

In real systems, schedules often include incomplete transactions (active transactions still running).
To simplify analysis, the **committed projection** C(S) of a schedule S includes only:

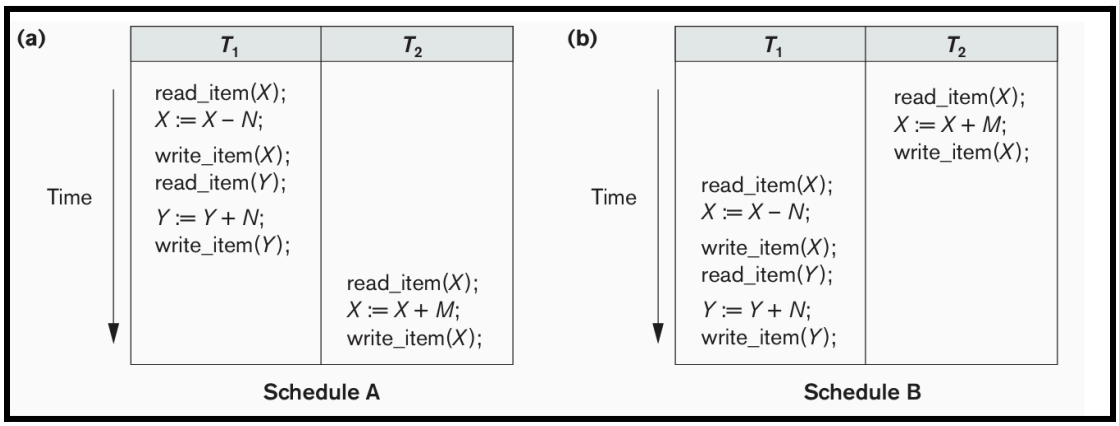
1. Operations from **committed transactions** (those with a ci).
2. Discards operations from incomplete or aborted transactions.

CHARACTERIZING SCHEDULES BASED ON SERIALIZABILITY

Serial Schedules

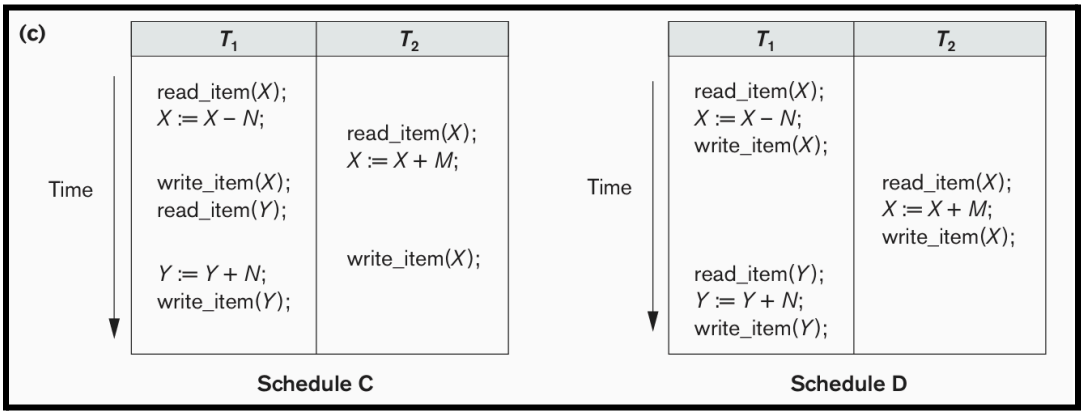
- A **serial schedule** is one in which all the operations of one transaction are executed **completely in sequence** before starting the next transaction.
- Example:
    - For transactions T1 and T2:
        - **Option 1**: Execute all operations of T1, then all operations of T2.
        - **Option 2**: Execute all operations of T2, then all operations of T1.

These are shown as **serial schedules** in Figure 21.5(a) and 21.5(b).



Schedules with Interleaving (non serial)

- When the DBMS allows **interleaving** of operations from **multiple transactions, the order of execution can vary. (non serial schedules)**
- Example: Operations of T1 and T2 may overlap in various ways, as shown in Figure 21.5(c).



What is Serializability?

- **Serializability** is the concept used to determine whether a schedule with interleaving of operations is **correct**.
- A schedule is **serializable** if it produces the **same result** as one of the **serial schedules** (either T1 followed by T2, or T2 followed by T1).

Why is Serializability Important?

- It ensures that the **interleaving of operations in concurrent transactions** does not violate the **consistency of the database.**
- This allows the DBMS to **execute transactions efficiently** while **maintaining the correctness** of the final results.

SERIAL, NONSERIAL, AND CONFLICT-SERIALIZABLE SCHEDULES

1. Serial Schedules

- **Definition**:
  - In a **serial schedule**, the operations of each transaction are executed **consecutively without any interleaving with other transactions.**
  - For T1 and T2, the schedule could be:
    - T1 is executed completely, followed by T2, or
    - T2 is executed completely, followed by T1.
- **Examples**:

    - Schedule A (Figure 21.5a): T1 → T2.
    - Schedule B (Figure 21.5b): T2 → T1. ***(above diagrams)***
- **Properties**:
    - Serial schedules are **always correct** because **no interleaving occurs.**
    - However, they are **inefficient as transactions must wait for others to finish.**

2. Non-serial Schedules

- **Definition**:
    - In a **nonserial schedule**, the operations of **multiple transactions** are **interleaved**.
    - This **improves concurrency** but **can lead to errors** if the **interleaving is not controlled properly.**
- **Examples**:
    - Schedule C and Schedule D (Figure 21.5c) are nonserial schedules.
- **Correctness**:
    - Some nonserial schedules are **correct** (e.g., Schedule D), while others are not (e.g., Schedule C).
    - To determine correctness, the concept of **serializability** is used. (***above diagrams)***

3. Serializable Schedules

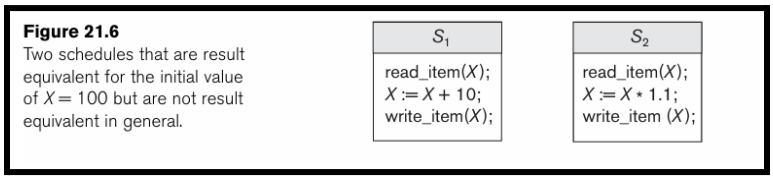
- **Definition**:
    - A nonserial schedule is **serializable** if it is **equivalent to a serial schedule** of the **same transactions.**
    - Serializable schedules **ensure correctness** by producing the **same results as a serial schedule.**
- **Conflict Serializability**:
    - A **schedule** is **conflict serializable** if it can be transformed into a **serial schedule** (where transactions execute one after the other) by **reordering non-conflicting operations**. This ensures the schedule produces the same result as a serial schedule, maintaining database consistency.
    - **Conflicting operations**:
        - Belong to different transactions.
        - Access the same data item.
        - At least one operation is a write.

When Are Two Schedules Conflict Equivalent?

Two schedules are **conflict equivalent** if:

1. They have the **same operations** (no operations are added or removed).
2. The **order of conflicting operations** is the **same** in both schedules.

Thus, Result, order, operations should be even in both, not just the result.



Examples

1. **Schedule D**:
    - Equivalent to Serial Schedule A (T1 → T2).
    - Nonconflicting operations r1(Y) and w1(Y) can be reordered to form the serial schedule.
    - Hence, Schedule D is **conflict serializable**. They are
    - **Result Equivalent. (only the result is same regardless the order)**
2. **Schedule C**:
    - Cannot be transformed into a serial schedule due to conflicts:
        - r2(X) and w1(X) conflict, so r2(X) cannot be reordered below w1(X).
        - w1(X) and w2(X) conflict, so w1(X) cannot be reordered below w2(X).
    - Hence, Schedule C is **not serializable**.

Why Serializability Matters

- Serializability ensures that interleaved execution of transactions produces **correct results**, just as if the transactions had executed serially.
- It allows for better **concurrency** while maintaining database consistency.

TESTING FOR CONFLICT SERIALIZABILITY

Conflict serializability checks if a given schedule can be rearranged into a serial schedule by reordering **non-conflicting operations**. This is determined using a **precedence graph** (or **serialization graph**).

Algorithm for Testing Conflict Serializability

1. **Create Nodes for Transactions**:
    - For every transaction Ti in the schedule, create a node Ti in the precedence graph.
2. **Add Edges for Conflicting Operations**:
    - For each pair of conflicting operations in the schedule:
        - If Ti performs a **write_item(X)** before Tj performs a **read_item(X)**, add an edge from Ti to Tj (Ti→Tj).
        - If Ti performs a **read_item(X)** before Tj performs a **write_item(X)**, add an edge from Ti to Tj (Ti→Tj).
        - If Ti performs a **write_item(X)** before Tj performs another **write_item(X)**, add an edge from Ti to Tj (Ti→Tj).
3. **Check for Cycles**:
    - If the precedence graph contains a **cycle**, the schedule is **not conflict serializable**.
    - If there are **no cycles**, the schedule is **conflict serializable**, and the graph determines the equivalent serial schedule.

Algorithm 21.1. Testing Conflict Serializability of a Schedule S

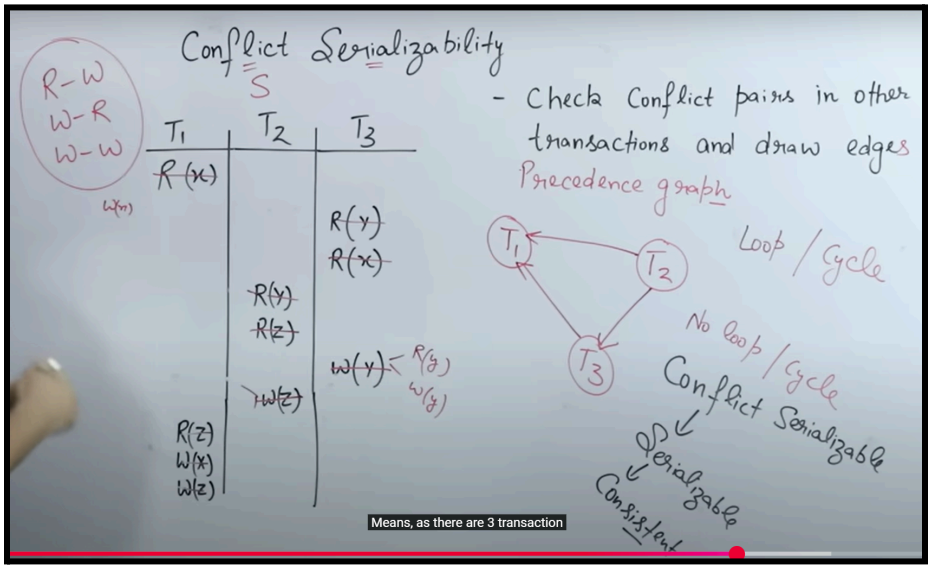
1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

Precedence Graph

A **precedence graph** is a **directed graph** where:

- **Nodes** represent **transactions.**
- **Edges** represent **dependencies between transactions** due to conflicting operations.

Example



- Before solving a problem we should know that the conflict pairs are:
- **R-W**
- **W-R**
- **W-W**
- Periodically check the table and draw edges in the precedence graph if a conflict occurs.

- First we check **T1, R(x)**. The conflicting partner is **W(x).** check if W(x) is present in other transactions. It does not exist, so there is no edge required in the graph.
- Next is **R(y)** in T3. No **W(Y**) is present in other transactions so no edge is drawn.
- Next is **R(x)**, **W(x)** is present in T1 so we draw an **edge** from **T3 to T1.**
- Repeat this process.
- **Remember if it is a write case, there are two things to check W-R as well as W-W. for example, in W(y) in T1, we have to check if W(Y) is present as well as W(R) is present or not in others. Also if you get an edge already, no need to overdraw another for the same.**
- **No loop i**s formed so it is **conflict serializable.**
- **How to check if it is just serializable? Find indegree and write indeg(0) first, delete the node. Do the same for other nodes. For the given example, the order is, T2->T3->T1.**
