UNIT II

CONCURRENCY CONTROL

Introduction to Concurrency Control

- Concurrency control in a DBMS ensures that multiple transactions can execute at the same time without causing data inconsistency or errors.
- This is important in a multi-user environment where many users access and modify the database simultaneously.
- It is necessary to maintain data integrity and consistency in multi-user environments. The key challenges of concurrency control include maintaining isolation among transactions, preventing lost updates, avoiding dirty reads, and ensuring serializability.
- Without concurrency control, issues such as race conditions, inconsistent retrievals, and anomalies can arise, leading to erroneous data.
- Concurrency control mechanisms provide solutions to the above problems by implementing techniques that regulate the execution of concurrent transactions and ensure that transactions adhere to the ACID (Atomicity, Consistency, Isolation, and Durability) properties of database transactions.

For example, if we take ATM machines and do not use concurrency, multiple persons cannot draw money at a time in different places. This is where we need concurrency.

Advantages

The advantages of concurrency control are as follows −

- Waiting time will be decreased.
- Response time will decrease.
- Resource utilization will increase.
- System performance & Efficiency is increased.

Why is Concurrency Control Important?

- **To maintain database consistency:** Ensure that the result of concurrent transactions is the same as if they were executed sequentially.
- **To prevent** conflicts caused by overlapping transactions.
- **To ensure isolation:** Each transaction behaves as if it is the only one operating on the database.

Common Problems in Concurrency

1. **Lost Updates**:
   - Occurs when two transactions update the same data, and one update overwrites the other.
   - **Example:**
     - Transaction T1 reads balance = $100 and subtracts $10 (new balance = $90).
     - Transaction T2 reads balance = $100 and subtracts $20 (new balance = $80).
     - T1 and T2 write back their updates, and the final balance becomes $80, losing T1's update.
2. **Dirty Reads**:
   - Happens when a transaction reads data that is not yet committed by another transaction.
   - **Example:**
     - T1 updates balance = $100 to $90 but has not committed.
     - T2 reads balance = $90 and uses this incorrect value, even if T1 eventually rolls back.
3. **Uncommitted Dependency**:
   - A transaction depends on changes made by another transaction that is not yet committed.
4. **Inconsistent Analysis**:
   - Occurs when a transaction reads data inconsistently because another transaction modifies part of the data during the read.
   - **Example:**
     - T1 calculates the total balance of two accounts A and B.
     - While T1 is reading, T2 transfers money from A to B. T1's result will be inconsistent.

Lock-Based Protocols

- Locking protocols in a Database Management System (DBMS) are mechanisms used to control access to data items to maintain **consistency, integrity, and concurrency** in transactions.
- A **locking protocol** is a set of rules used in a Database Management System (DBMS) to manage access to data in a way that ensures **concurrent transactions** do not cause inconsistencies.
- They prevent **lost updates, dirty reads, uncommitted data conflicts and Inconsistent Analysis** by ensuring that multiple transactions do not interfere with each other in an undesirable manner.
- It helps in maintaining the ACID properties of a transaction.

Locks:

Locks are mechanisms to control access to database items. A transaction must acquire a lock on a data item before reading or writing it. This ensures no other transaction modifies the data while the lock is held.

Types of Locks

1. **Shared Lock (S)**:
    - Multiple transactions can acquire a shared lock on the same data item for reading.
    - No transaction can modify the data while it is shared-locked.
2. **Exclusive Lock (X)**:
    - Only one transaction can acquire an exclusive lock on a data item for writing.
    - No other transaction can read or write the data while it is exclusive-locked.

Locking Protocols

Types of Locking Protocols

There are **four main types** of locking protocols:

1. **Simplistic Locking Protocol**
2. **Pre-emptive Locking Protocol**
3. **Two-Phase Locking (2PL) Protocol**
4. **Timestamp-Based Protocol**

1. Simplistic Locking Protocol

- This is the most basic locking protocol.
- A transaction **locks** a data item before performing operations on it.
- The lock is **released only after** the transaction is completed.

Example:

A user locks a file before editing and unlocks it after saving the changes.

2. Preemptive Locking Protocol

- In this protocol, a transaction locks **all the required data items** before it begins execution.
- If a transaction cannot lock all required data items, it waits until all are available.

Example:

Before booking a flight ticket, the system locks the available seat to prevent others from booking it simultaneously.

3. Two-Phase Locking (2PL) Protocol

This is one of the most commonly used locking protocols. It ensures **serializability** of transactions.

Two phases of locking:

1. **Growing Phase**: A transaction can only acquire locks and **cannot release** any lock.
2. **Shrinking Phase**: A transaction can only release locks and **cannot acquire** any new lock.

Example:

- Transaction **T1** wants to read and write a data item **A**.
- It first **acquires a lock** on **A** (Growing phase).
- It performs read/write operations on **A**.
- It **releases the lock** after completing the operation (Shrinking phase).

a) **Strict Two-Phase Locking Protocol:**
   a. Exclusive locks are held until the transaction commits or aborts.
   b. Prevents **dirty reads** because changes are visible only after the transaction is completed.

b) **Rigorous Two-Phase Locking Protocol:**
   a. Both shared and exclusive locks are held until the transaction commits.
   b. Provides stronger guarantees but may lead to more waiting.

4. Timestamp-Based Protocol

- Uses a **timestamp** to determine the order of transactions.
- Each transaction gets a unique timestamp **when it starts**.
- Older transactions get priority over newer transactions.
- Avoids deadlocks.

Example:

- Suppose **T1** starts at **10:00 AM**, and **T2** starts at **10:05 AM**.
- If **T2** wants to access the same data as **T1**, it must wait until **T1** finishes.

Deadlock Handling

What is Deadlock in DBMS?

- A deadlock in DBMS is an undesired state that occurs when a process waits indefinitely for a resource that is being held by another process.
- To better understand the concept of deadlock, take a transaction T1 that has a lock on a few rows in the table Employee and has to change certain rows in another table Salary.
- There is also another transaction T2 that has a lock on the Salary table and needs to update a few rows in the Employee table, which is already held by transaction T1.
- In this case, both transactions and processes are waiting for each other to release the lock, and the processes are waiting for each other to release the resources. As a result of the preceding scenario, none of the jobs are accomplished, which is referred to as deadlock.

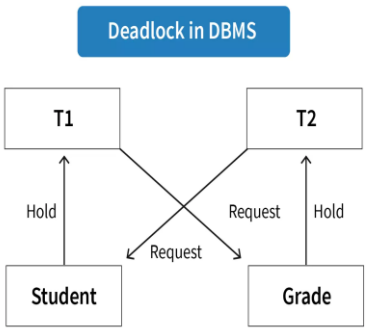
Why does Deadlock occur in DBMS?

- A deadlock arises when two transactions wait endlessly for each other to unlock data.
- A deadlock arises when two transactions, P1 and P2, exist in the following mode:

  T1 = access data items **Student** and **Grade**

  T2 = access data items **Grade** and **Student**

  - T2 cannot begin if t1 has not unlocked data item **Grade**, and T1 cannot continue if T2 has not unlocked data item **Student**.
  - As a result, T1 and T2 each wait for the other to unlock the desired data item. This type of deadlock is also described as a deadly embrace. The table below shows how a deadlock condition occurs in DBMS.



- A deadlock occurs when two or more transactions wait indefinitely for resources locked by each other.
- Deadlocks can severely impact system performance.

Deadlock Handling in DBMS

There are three traditional ways to handle deadlock in DBMS:

1. **Deadlock prevention.**
2. **Deadlock avoidance.**
3. **Deadlock detection and removal.**

All three ways can be used in both centralized and distributed database systems to handle the deadlock condition.

1. Deadlock Prevention

- The deadlock prevention strategy prohibits any transaction from acquiring locks that will result in deadlocks. According to the convention, when multiple transactions request to lock the same data item, only one of them is granted the lock.
- Pre-acquisition of all locks is one of the most popular deadlock prevention strategies. In this approach, a transaction gets all locks before beginning to execute and keeps them for the duration of the transaction.
- If another transaction requires any of the previously acquired locks, it must wait until all of the locks required are accessible. This strategy prevents the system from becoming jammed because none of the waiting transactions are holding any locks.

Deadlock Prevention Techniques:

1. **Wait-Die Scheme**: Older transactions wait for younger transactions to release locks. Younger transactions are aborted.
2. **Wound-Wait Scheme**: Older transactions force younger ones to release locks. Younger transactions are aborted.

	Wait/Die (Non-preemptive technique)	**Wound/Wait (preemptive technique)**
The older process requires a resource held by the younger process.	The older process waits.	The younger process dies.
The younger process requires a resource held by the older process.	The older process dies.	The younger process waits.

For example, assume there are two transactions, T1 and T2, in which T1 attempts to lock a data item that is already locked by T2. Here are the algorithms:

- **Wait-Die** − T1 is allowed to wait if he is older than T2. If T1 is younger than T2, the process is aborted and then restarted.
- **Wound-Wait** − T2 is cancelled and restarted if T1 is older than T2. T1 is allowed to wait if it is younger than T2.

3. Deadlock Avoidance

- Stopping Deadlocks before they occur is often better than recovering from them.
- Deadlock avoidance is an example of careful resource allocation.
- The DBMS checks whether granting a resource to a transaction will potentially lead to a Deadlock. If it predicts a Deadlock, it denies the request, making the system safe.
- Another method for preventing Deadlock is Banker's Algorithm, which checks whether resources can be allocated safely so that no Deadlock will occur. For the sake of practical implementation, Deadlock avoidance may involve transactions to make decisions that guarantee no circular wait conditions.
- However, Deadlock avoidance may degrade the performance of the system since the algorithms running for continuous monitoring and forecasting possible Deadlocks incur computational overhead.

4. Deadlock detection and Removal

- In deadlock detection the lock management technique checks the **wait-for-graph for cycles** on a regular basis to detect deadlocks.
- If a cycle is found in the graph, it indicates a deadlock

Deadlock detection methods:

DBMS uses different approaches to detect deadlocks:

A. Wait-for Graph (WFG)

A **Wait-for Graph (WFG)** is a directed graph where:

- Nodes represent transactions.
- An edge from Transaction **T1 → T2** indicates that **T1** is waiting for a resource held by **T2**.
- A **cycle** in the graph indicates a **deadlock**.

B. Detection Mechanism:

The system periodically checks the **Wait-for Graph** for cycles using algorithms like:

- **Depth First Search (DFS)**
- **Tarjan's Strongly Connected Components (SCC) Algorithm**

C. Timeout-Based Detection

- If a transaction waits beyond a predefined time, the system assumes a deadlock and **rolls back** the transaction.
- This method is **simple but can lead to false deadlock detection** if the timeout is too short.

2. Deadlock Removal Methods

Once a deadlock is detected, the system must **break the cycle** using **one of the following approaches**:

A. Transaction Rollback (Victim Selection)

- The system **chooses a transaction to roll back** to release its locks and break the cycle.
- **Victim selection criteria:**
    - **Transaction Priority** – Lower-priority transactions are rolled back.
    - **Least Progress** – Transactions that have made less progress are aborted.
    - **Lowest Resource Usage** – Transactions holding fewer resources are preferred for rollback.

B. Preemption (Forcefully Releasing Locks)

- The system **temporarily suspends a transaction** and **releases its locks**.
- Other transactions proceed, and the suspended transaction restarts later.

C. Kill All Transactions in Deadlock

- In extreme cases, the system **rolls back all deadlocked transactions** and restarts them.

Multiple Granularity

- Multiple granularity is a locking mechanism in database concurrency control that allows different levels of data items (such as tables, rows, or even attributes) to be locked at different granularities.
- This technique enhances efficiency by reducing the number of locks required and improving concurrency.
- Multiple granularity allows locking at different levels of the database, such as tables, rows, or even fields.
- This increases efficiency by reducing the number of locks required.

Types of Locks:

- **Intention Locks** (indicate future locking intention at finer granularity):
  - **Intention Shared (IS)**: Intends to set shared locks at lower levels.
  - **Intention Exclusive (IX)**: Intends to set exclusive locks at lower levels.
  - **Shared-Intention Exclusive (SIX)**: Shared lock at the current level but intends exclusive locks at lower levels.
- **Data Locks:**
  - **Shared (S)**: Allows multiple transactions to read but not write.
  - **Exclusive (X)**: Allows only one transaction to read and write.

- **Compatibility Matrix** is used in database management systems (DBMS) to determine whether multiple transactions can hold locks on the same data item simultaneously.
- By using the **Compatibility Matrix** it easy to implement the mechanism of multiple granularity

Example of Multiple Granularity Locking

Let's break it down step by step with a **real-world example** using a **university database**.

Scenario:

Imagine a **university database** that contains student records. A **professor** wants to read **a specific student's grade** from the database. Meanwhile, the **administration** wants to update student records.

Hierarchy of Database Objects:

1. **University Database (DB)**
2. **Department Table (T) – "Computer Science Students"**
3. **Page (P) – A page storing student records**
4. **Record (R) – A single student's record**

Step-by-Step Locking Process

Transaction 1: Professor Reading a Student's Grade

- The professor wants to read a **student's grade**, so they need a **Shared (S) lock** on that student's record.
- **Locking hierarchy applied:**
    1. **Intention Shared (IS) Lock on Database (DB)** – Intends to acquire shared locks at a lower level.
    2. **IS Lock on Table (T) – "Computer Science Students"** – Shows intention to read data
    3. **IS Lock on Page (P)** – The page containing student records
    4. **Shared (S) Lock on Student Record (R)** – The professor reads the student's grade.

This allows multiple professors to read different student records at the same time.

Transaction 2: Admin Updating a Student's Record

- The admin wants to update the student's **address**, requiring an **Exclusive (X) lock** on that record.
- **Locking hierarchy applied:**
    1. **Intention Exclusive (IX) Lock on Database (DB)** – Intends to make changes at a lower level.
    2. **IX Lock on Table (T) – "Computer Science Students"** – Shows intention to update data.
    3. **IX Lock on Page (P)** – The page containing student records.
    4. **Exclusive (X) Lock on Student Record (R)** – The admin updates the record.

Since an **Exclusive (X) lock conflicts with a Shared (S) lock**, the admin must wait until the professor finishes reading.

Lock Compatibility Table

Transaction 1 (Professor - Read S Lock)	**Transaction 2 (Admin - Update X Lock)**	**Conflict?**
Shared (S) Lock on Record (R)	**Exclusive (X) Lock on Record (R)**	**Yes** (Admin must wait)

Snapshot Isolation in DBMS

- Snapshot Isolation (SI) is a concurrency control mechanism used in database management systems (DBMS) to ensure that transactions work with a consistent snapshot of the database.
- It prevents issues like **dirty reads** and **non-repeatable reads**, but it does not guarantee **serializability** in all cases.

How Snapshot Isolation Works

- When a transaction starts, it gets a **snapshot** of the database.
- It operates on that snapshot rather than the latest committed state of the database.
- Any changes made by other transactions after this snapshot are **not visible** to the current transaction.
- If a transaction attempts to modify a row that another transaction has changed after its snapshot, it **fails** due to a write-write conflict.

Example: Bank Account Transactions

Imagine a bank account with ₹1000 in it.

Account_ID	Balance
101	₹1000

Two Users Performing Transactions at the Same Time

1. **User A withdraws ₹200:**
    - Takes a snapshot of the account (₹1000).
    - Updates balance to ₹800.
    - Waiting to save (commit) the change.
2. **User B deposits ₹500:**
    - Also takes a snapshot (₹1000).
    - Updates balance to ₹1500.
    - Waiting to save (commit) the change.
3. **Now, both transactions try to commit:**
    - If **User A commits first**, the new balance in the database is ₹800.
    - When **User B tries to commit**, it will **fail** because it used an old snapshot (₹1000 instead of ₹800).
    - **User B must restart** and take a fresh snapshot.

Recovery System

Introduction

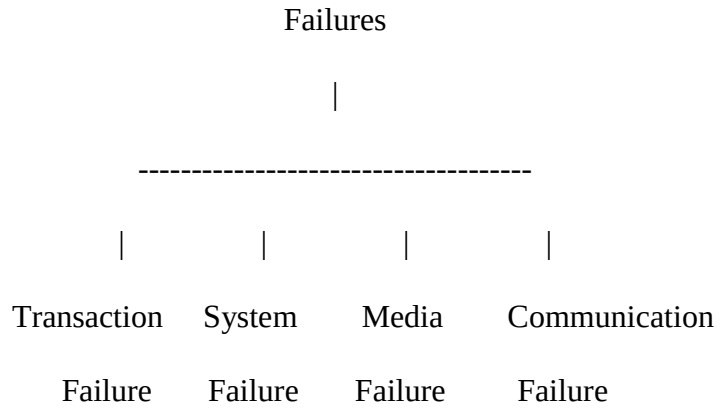
- In a Database Management System (DBMS), a **Recovery System** ensures that the database remains consistent even when failures occur.
- Failures can happen for various reasons, such as hardware issues, software bugs, or power outages.
- Recovery mechanisms restore the database to its last consistent state and uphold the **ACID properties** (Atomicity, Consistency, Isolation, and Durability).

Why is a Recovery System Important?

1. **Prevents Data Loss**: Ensures that no data is lost during unexpected failures.
2. **Maintains Consistency**: Keeps the database in a valid state.
3. **Ensures Reliability**: Protects the database's integrity and ensures reliable operations.
4. **Handles Failures Gracefully**: Automatically recovers from failures with minimal user intervention.

Failure Classification

Failures in a DBMS can be categorized into the following types:



1. Transaction Failures

These occur when a transaction cannot be completed successfully. Reasons include:

- **Logical Errors**: Errors in the transaction code, such as division by zero or invalid input.
- **System Errors**: Situations like deadlocks or lack of resources, which force the transaction to stop.

Example:

Suppose a transaction is transferring $500 from Account A to Account B. If there is a power outage after debiting Account **A** but before crediting Account **B**, the transaction fails.

2. System Failures

Happen when the entire system crashes due to hardware or software issues. The system stops working temporarily, and the database resides in memory, which may be lost.

Example Causes:

- Power failures.
- Operating system crashes.

3. Media Failures

Involve physical damage to the storage media (e.g., disk crashes or corruption), causing permanent data loss.

Example:

If a hard disk containing the database files fails, the entire database may become inaccessible.

4. Communication Failures

Occur in distributed databases when there is a network failure during a transaction.

Example:

A distributed banking system loses network connectivity while updating accounts in different branches.

Storage in Recovery Systems

The recovery system relies on various storage mechanisms to ensure data safety and recovery.

Types of Storage

1. **Volatile Storage**:
    - Example: RAM (Random Access Memory).
    - Data is lost when power is off.
2. **Non-Volatile Storage**:
    - Example: Hard drives, SSDs (Solid State Drives).
    - Used for long-term data storage.
3. **Stable Storage**:
    - A hypothetical type of storage where data is never lost.
    - Achieved through redundancy (e.g., replication across multiple disks).

Storage Structures

1. **Log Files**: Record every transaction's details, such as changes made to the database.
2. **Data Files**: Contain the actual data stored in the database.
3. **Checkpoint**: A snapshot of the database taken periodically. It reduces the recovery time by limiting the number of log records to be processed during recovery.

Recovery and Atomicity

What is Atomicity?

Atomicity ensures that a transaction is treated as a single unit. It either completes fully or is rolled back entirely.

How is Atomicity Ensured?

1. **Undo Operations**:
    - Reverse the changes made by incomplete or failed transactions.
    - Ensures that partially completed transactions do not affect the database.
2. **Redo Operations**:
    - Reapply changes made by committed transactions to ensure durability.

Example:

Imagine a transaction transferring $500 from Account A to Account B:

- Step 1: Debit $500 from Account A.
- Step 2: Credit $500 to Account B.

If the system crashes after Step 1 but before Step 2, the recovery system will undo Step 1 to maintain atomicity.

Recovery Algorithms

Recovery algorithms are procedures to restore the database to a consistent state after a failure.

1. Log-Based Recovery

- Maintains a **log file** recording every transaction's details.
- Types of log records:
    1. <T, Start>: Indicates the start of transaction T.
    2. <T, X, Old_Value, New_Value>: Transaction T changes value of X from Old_Value to New_Value.
    3. <T, Commit>: Indicates the successful completion of transaction T.

Process:

1. **Redo** all committed transactions.
2. **Undo** all uncommitted transactions.

Example:

If the log contains the following:

<T1, Start>

<T1, A, 1000, 500>

<T1, Commit>

<T2, Start>

<T2, B, 200, 300>

During recovery:

- **Redo** changes made by T1 (as it is committed).
- **Undo** changes made by T2 (as it is not committed).

2. Checkpoint-Based Recovery

- Saves the database state at regular intervals (checkpoints).
- Reduces recovery time by ignoring logs before the last checkpoint.

Process:

1. Identify the last checkpoint.
2. Undo all uncommitted transactions after the checkpoint.
3. Redo all committed transactions after the checkpoint.

Example:

Checkpoint at time t1, and logs show:

Checkpoint at t1

<T1, Commit>

<T2, Start>

<T2, A, 300, 400>

During recovery:

- Ignore logs before t1.
- Redo T1 and undo T2.

Diagram: Timeline showing checkpoints and associated logs.

3. Shadow Paging

- Maintains two copies of the database:
    1. **Current Page Table**: Used for ongoing transactions.
    2. **Shadow Page Table**: Stable and unaltered copy of the database.

Process:

1. During a transaction, changes are made to the current page table.
2. When the transaction commits, the shadow page table is updated.
3. If the transaction fails, the current page table is discarded, and the shadow table is used.

Advantages:

- No need for logs.

Disadvantages:

- High storage overhead.
- Difficult to implement in distributed systems.