Concurrency Control:

Concurrency control in a Database Management System (DBMS) is crucial to ensure that multiple transactions (SQL statements) can execute simultaneously while maintaining data consistency and preventing conflicts. In this explanation, I'll provide detailed information about concurrency control in DBMS with an example.

Example Scenario:

Consider a simple banking application with two SQL transactions:

Transaction 1 (T1): A customer wants to transfer $100 from their savings account (SA1) to their checking account (CA1).

BEGIN TRANSACTION;

UPDATE savings_accounts SET balance = balance - 100 WHERE account_id = 'SA1';

UPDATE checking_accounts SET balance = balance + 100 WHERE account_id = 'CA1';

COMMIT;

Transaction 2 (T2): A customer wants to check their savings account balance.

BEGIN TRANSACTION;

SELECT balance FROM savings_accounts WHERE account_id = 'SA1';

COMMIT;

Conflicts of Concurrency:

Concurrency in a database system, where multiple transactions execute simultaneously, can lead to various conflicts due to the overlapping or simultaneous execution of these transactions. These conflicts can potentially result in data inconsistencies, loss of updates, or incorrect results. There are four primary types of conflicts in concurrency:

1.)Read-Write Conflict (or Write-Read Conflict):

Occurs when one transaction reads a data item that another transaction is in the process of modifying.

If a transaction reads a data item while another transaction is modifying it, the reader may see an inconsistent or partially updated value.

This type of conflict is typically resolved using locks. The transaction that writes data acquires a write lock, preventing other transactions from reading or writing the same data item until the lock is released.

2.)Write-Write Conflict:

Occurs when two or more transactions attempt to modify the same data item simultaneously.

This situation can lead to lost updates, where one transaction's changes overwrite another's changes, causing data inconsistency.

Write-write conflicts are typically resolved using write locks. A transaction must acquire an exclusive write lock before modifying a data item.

Concurrency Control Techniques:

1. **Lock Based Control**

Lock-based concurrency control is a technique in which transactions lock data items to control access, ensuring that only one transaction can modify a particular data item at a time while allowing multiple transactions to read it. In this approach, conflicts are resolved by blocking (waiting) or aborting (rolling back) transactions, depending on the type of lock requested and the locks already held by other transactions. Let's explore lock-based concurrency control with an example:

In a bank database, we have two transactions:

Transaction 1 (T1): A customer wants to transfer $500 from their savings account (SA1) to their checking account (CA1).

```
BEGIN TRANSACTION;

SELECT balance FROM savings_accounts WHERE account_id = 'SA1';

-- Check if there is enough balance

IF balance >= 500 THEN

   -- Deduct from savings

   UPDATE savings_accounts SET balance = balance - 500 WHERE
account_id = 'SA1';

   -- Add to checking

   UPDATE checking_accounts SET balance = balance + 500 WHERE
account_id = 'CA1';

END IF;

COMMIT;
```

Transaction 2 (T2): A customer wants to check the balance of their savings account (SA1).

```
BEGIN TRANSACTION;

SELECT balance FROM savings_accounts WHERE account_id = 'SA1';

COMMIT;
```

Now, let's break down how these transactions work and how lock-based concurrency control manages them:

Transaction 1 (Fund Transfer - T1):

- T1 begins by starting a transaction (BEGIN TRANSACTION).
- It then tries to read the balance of the savings account ('SA1') by executing SELECT balance FROM savings_accounts WHERE account_id = 'SA1';.
- When this SELECT operation is performed, T1 acquires a read lock on the 'SA1' data item. This lock ensures that other transactions won't be able to modify 'SA1' while T1 is reading it.
- T1 checks if there is enough balance for the transfer (i.e., if balance >= 500), and if so, it proceeds to update the accounts. It acquires write locks on 'SA1' and 'CA1' during these update operations.
- After the operations are complete, T1 commits the transaction (COMMIT).

Transaction 2 (Balance Inquiry - T2):

- T2 also starts a transaction (BEGIN TRANSACTION).
- It attempts to read the balance of the savings account ('SA1') by executing SELECT balance FROM savings_accounts WHERE account_id = 'SA1';.
- Because T1 currently holds a read lock on 'SA1' (acquired during its SELECT operation), T2 must wait for T1 to release the lock.
- Once T1 commits and releases the read lock on 'SA1', T2 can proceed and acquire its own read lock to check the balance.

- T2 then commits its transaction (COMMIT).

Key Points:

- Lock-based concurrency control ensures that multiple transactions accessing the same data item do not interfere with each other.
- In this example, T1 and T2 both want to read the balance of 'SA1,' but T2 has to wait until T1 releases the read lock.
- T1, upon committing, releases the read lock on 'SA1,' allowing T2 to acquire it and proceed with the balance inquiry.

2. Timestamp based control:

Timestamp-based concurrency control is a technique used in Database Management Systems (DBMS) to manage concurrent transactions based on their timestamps. Each transaction is assigned a unique timestamp representing the time it began. Timestamp control allows for ordering transactions based on their timestamps and enforces a serializability order that ensures the transactions execute in a way that maintains data consistency. Let's explore this technique with an example in the context of a bank database system.

Example Scenario:

In a bank database, we have two transactions:

Transaction 1 (T1): A customer wants to transfer $500 from their savings account (SA1) to their checking account (CA1).

BEGIN TRANSACTION;

SELECT balance FROM savings_accounts WHERE account_id = 'SA1';

-- Check if there is enough balance

IF balance >= 500 THEN

-- Deduct from savings

UPDATE savings_accounts SET balance = balance - 500 WHERE account_id = 'SA1';

-- Add to checking

UPDATE checking_accounts SET balance = balance + 500 WHERE account_id = 'CA1';

END IF;

COMMIT;

Transaction 2 (T2): A customer wants to check the balance of their savings account (SA1).

BEGIN TRANSACTION;

SELECT balance FROM savings_accounts WHERE account_id = 'SA1';

COMMIT;

Transaction 1 (T1): Fund Transfer

Transaction 1, or T1, begins as a customer initiates a fund transfer. The process starts with T1 by selecting the balance of the customer's

savings account, referred to as 'SA1.' T1 checks whether there is a sufficient balance for the requested transfer, specifically verifying if there are at least $500 in the savings account. If the balance condition is met, T1 proceeds to deduct the amount from the savings account and simultaneously adds it to the checking account, known as 'CA1.' Throughout this process, T1 is keen on maintaining the integrity of the database.

Transaction 2 (T2): Balance Inquiry

Transaction 2, or T2, represents a balance inquiry made by another customer on their savings account 'SA1.' Similar to T1, T2 begins its operation by selecting the balance of 'SA1.' This allows the customer to inquire about their account balance.

In the world of concurrent transactions, issues of data consistency and integrity often arise. It is here that timestamp-based concurrency control comes into play. Each transaction is assigned a unique timestamp, reflecting the time at which it began. In this example, we assume that T1 starts before T2, so it is assigned a lower timestamp than T2.

These timestamps serve as the determinants for the execution order of the transactions. T1, with its earlier timestamp, is permitted to complete its operations before T2 can proceed. As a result, T1 performs the fund transfer and updates the accounts accordingly, all while adhering to the constraints and business rules.

Meanwhile, T2 patiently awaits the release of the necessary resources (in this case, the 'SA1' account) held by T1. Once T1 commits its transaction, it releases the resources it held, allowing T2 to execute its balance inquiry. In this way, timestamp-based control guarantees that T2 does not read inconsistent or partially updated data but rather sees the finalized and accurate balance information.

In summary, timestamp-based concurrency control ensures that transactions in the bank database system execute in a predetermined and consistent order, preventing data inconsistencies and maintaining the integrity of the database. In this example, it ensures that the fund transfer (T1) is completed before the balance inquiry (T2), safeguarding data correctness and reliability in the system.