# Chapter 8

# Transaction Processing Concepts

**Objectives**

Transaction is a logical unit of related database operations executed on the server. For complex applications, there may be multiple transactions at one time executing concurrently. In this chapter, students will learn about the concepts of Transaction, Schedules and Serializability etc.

**Structure**

- Transaction
- ACID Properties and Transaction States
- Transaction States
- Schedules
- Serializability
- Conflict and View Serializable Schedule

## 8.0 INTRODUCTION

There are two basic types of database management systems – Single user systems and Multi user systems. In single user systems as the name indicates, only one user uses the system, whereas in Multi user systems, many users may use the same system at the same time and access the system concurrently. Almost every database system now a days are multi user systems. Railway Reservation System is an example of multi user system. In these type systems, multiple users can send the request to the server and it is the responsibility of the database management system to handle these requests in a systematic way, known as Transaction Management.

A transaction can be defined as a sequence of operations performed together as a single logical unit of work. It is a single logical unit of work which evolves one or more database access operations.

In simple terms:

*A transaction is defined as a logical unit of work which involves a sequence of steps but which normally will be considered as one action and which preserves consistency of the database.*

The transaction is executed as a series of reads and writes of database objects are explained below:

**Read (X)** command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a main memory buffer (if it's not already there).
3. Copy item X from the buffer to the program variable named X.

**Write (X)** command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a main memory buffer (if it's not already there).
3. Copy item X from the program variable named X into its location in the buffer.
4. Store the updated block from the buffer back to disk.

Consider you are working on a system for a bank. A customer A wants to transfer an amount of Rs. 1000 to another customer B. This simply requires two steps:

i) Subtracting the money from the account A
ii) Adding the money to the account B

The code to create this transaction will require two updates to the database. For example, there will be two SQL statements: first update command to decrease the amount from the account A and second update command to increase the amount in the account B.

This transaction can be represented as follows:

| **Transaction T1** |
|---|
| UPDATE Account_A SET Balance= Balance -1000;<br><br>1. **Read**(A)<br>2. A = A – 100<br>3. **Write**(A)<br><br>UPDATE Account_B SET Balance= Balance +1000;<br><br>4. **Read**(B)<br>5. B = B + 100<br>6. **write**(B) |

**Table 8.1 An Example of Transaction**

**Commit Point of the Transaction**

A transaction T reaches its commit point when all its operations that access the database have been executed successfully and recorded to the log. Beyond the commit point, the transaction is said to be committed and its effects assumed to be permanently recorded in the database.

If an error occurs during the execution of one of the operation, transaction aborts and rolls back. For example while transferring an amount from Account A to Account, it completes the operation of withdrawing an amount from Account. However while crediting this amount to Account B a failure occurs. In this case, the transaction will abort and will rollback the first operation also.

## 8.1 ACID PROPERTIES OF TRANSACTION

A transaction must possess the four properties called ACID (Atomicity, Consistency, Isolation and Durability). These properties are explained as follows:

**Atomicity**
This property states that database modifications must follow an "*all or nothing*" rule. Each transaction is said to be "atomic." If one part of the transaction fails, the entire transaction fails. It is critical that the database management system maintain the atomic nature of transactions in spite of any DBMS, operating system or hardware failure.

The transaction subtracts Rs. 1000 from Account A and adds Rs.1000 to Account B. If it succeeds, it would be valid, because the data continues to satisfy the constraint. However, assume that after removing 1000 from A, the transaction is unable to modify B due to some failure. If the database retains A's new value, atomicity and the constraint would both be violated. Atomicity requires that both parts of this transaction complete or neither.

In the above example of transaction shown in Table 8.1, transaction may fail after step 3 and before step 6 (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.

**Consistency**
Consistency is a state in which all the data is in a consistent state after a transaction is completed successfully. It states that only valid data will be written to the database. If, for some reason, a transaction is executed that violates the database's consistency rules, the entire transaction will be rolled back and the database will be restored to a state consistent with those rules. On the other hand, if a transaction successfully executes, it will take the database from one state that is consistent with the rules to another state that is also consistent with the rules.

For instance, if funds are transferred between the two accounts, a withdrawal and deposit must both be committed to the database, so that the accounting system does not fall out of balance. In example shown in Table 8.1, the sum of A and B is unchanged by the execution of the transaction.

**Isolation**
Isolation states that any data modification made by a transaction must be isolated from the modifications made by any other concurrent transactions. In other words, a transaction either accesses data in the state it was on, before another concurrent transaction modified it or it access the data after the second transaction has been completed. There is no scope for the transaction to see an intermediate state.

To demonstrate isolation, we assume two transactions execute at the same time, each attempting to modify the same data. One of the two must wait until the other completes in order to maintain isolation.

Consider two transactions. $T_1$ transfers Rs. 1000 from A to B. $T_2$ transfers Rs.1000 from B to A. Combined, there are four actions:

- subtract 1000 from A

- add 1000 to B.
- subtract 1000 from B
- add 1000 to A.

If these operations are performed in order, isolation is maintained, although $T_2$ must wait. Consider what happens, if $T_1$ fails half-way through. The database eliminates $T_1$'s effects, and $T_2$ sees only valid data.

By interleaving the transactions, the actual order of actions might be: $A - 1000$, $B - 1000$, $B + 1000$, $A + 1000$. Again consider what happens, if $T_1$ fails. $T_1$ still subtracts 1000 from A. Now, $T_2$ adds 1000 to A restoring it to its initial value. Now $T_1$ fails. What should A's value be? $T_2$ has already changed it. Also, $T_1$ never changed B. $T_2$ subtracts 1000 from it. If $T_2$ is allowed to complete, B's value will be 1000 too low, and A's value will be unchanged, leaving an invalid database. This is known as a write-write failure, because two transactions attempted to write to the same data field.

**Durability**
Durability states that any changes in data by a completed transaction remain permanently in effect in the system. Hence any change in data due to a completed transaction persists even in the event of a system failure.

These properties mentioned above are summarized in Table 8.2.

| | **Property** | **Description** |
|---|---|---|
| A | Atomicity | If one part of the transaction fails, the entire transaction fails. "All or Nothing" |
| C | Consistency | All the data is in a consistent state after a transaction is completed successfully. Otherwise rolls back to the original state. |
| I | Isolation | Each transaction is executed as if it was the only one running. |
| D | Durability | Once committed, transaction is permanent and can not be rolled back. |

**Table 8.2: Brief Summary of the Transaction ACID Properties**

## 8.2 TRANSACTION STATES

In a database system a transaction might consist of one or more data-manipulation statements and queries, each reading and/or writing information in the database. Users of database systems consider consistency and integrity of data as highly important. A simple transaction is usually issued to the database system in a language like SQL wrapped in a transaction, using a pattern similar to the following:

1. Begin the transaction
2. Execute a set of data manipulations and/or queries
3. If no errors occur then commit the transaction and end it
4. If errors occur then rollback the transaction and end it

If no errors occurred during the execution of the transaction then the system commits the transaction. A transaction commit operation applies all data manipulations within the scope of the transaction and persists the results to the database. If an error occurs during the transaction, or if the user specifies a rollback operation, the data manipulations within the transaction are not persisted to the database. In no case can a partial transaction be committed to the database since that would leave the database in an inconsistent state.

These transaction states can be classified as follows:

1) ***Active***: This is the initial state. Transaction stays in this state while it is executing.

2) ***Partially Committed***: A database transaction enters this phase when its final statement has been executed. At this phase, the database transaction has finished its execution, but it is still possible for the transaction to be aborted because the output from the execution may remain residing temporarily in main memory - an event like hardware failure may erase the output.

3) ***Aborted***: If it is found that normal execution can be no longer performed then aborted is the state after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction. In other words, an aborted transaction must have no effect on the database, and thus any changes it made to the database have to be undone, or in technical terms, ***rolled back***. The database will return to its consistent state when the aborted transaction has been rolled back. The DBMS's recovery scheme is responsible to manage transaction aborts.

4) ***Committed***: A database transaction enters the committed state when enough information has been written to disk after completing its execution with success. In this state, so much information has been written to disk that the effects produced by the transaction cannot be undone via aborting; even when a system failure occurs, the changes made by the committed transaction can be re-created when the system restarts.

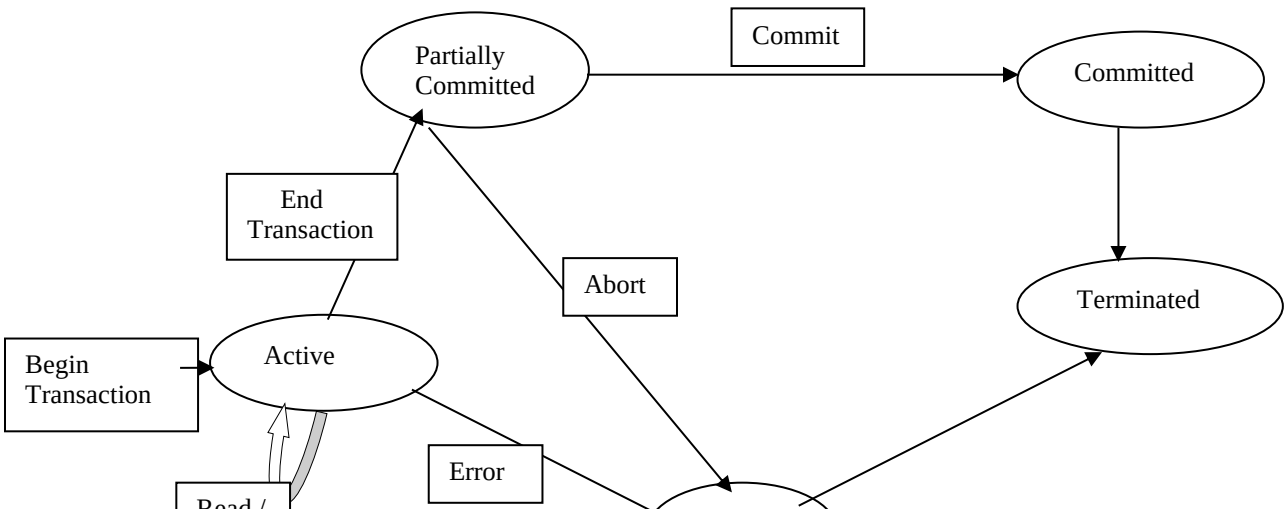

**Fig. 8.1 Transaction Process Diagram**

Figure 8.1 shows the state transaction diagram of a transaction execution. A transaction enters the active state at start. It remains in this state while issuing read and write operations. When the transaction has no further read or writes operations to perform, it enters the partially committed state. The concurrency control scheme at this stage performs the necessary checks to ensure that the transaction can be allowed to commit. Moreover, the recovery procedures required to protect against a system failure are taken. Once both of these operations are completed, the transaction enters the committed state. If any of the checks fail or the transaction cannot proceed due to any logical error, it enters the aborted state. The terminated state is reached when the transaction leaves the system.

A simple example of a transaction is a withdrawal from a bank account, which can be described as a set of actions that changes the state of an account balance (by reducing it). For this transaction, the system must execute a procedure that consists of three operations.

- Verify that a authorize customer is withdrawing the amount (Credential Check)
- Verify that whether sufficient amount is there to withdraw (Balance Check)
- Withdraw specified amount from the account (Withdraw)
- Update the record of the balance of the account (Balance Update)

Following is the state diagram, showing this transaction for withdrawal from the account.

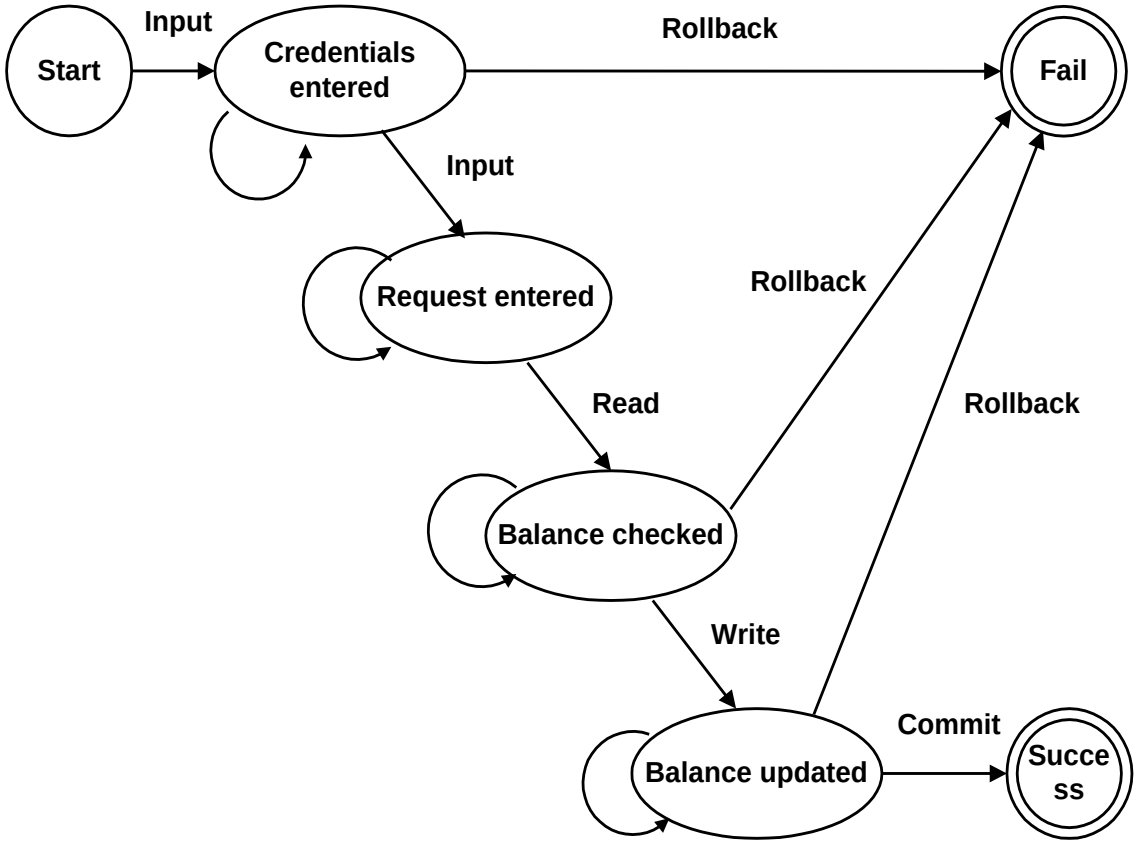


**Fig. 8.2 State Diagram for a Transaction of Cash Withdraw**

## 8.3 SCHEDULE

When several transactions are executing concurrently then the order of execution of various instructions is known as a schedule. For example, consider a schedule S which consists of transactions T1, T2, T3 ….T4. These transactions are in order of the operations in such a way that each transaction Ti that participates in S, the operations of the Ti in S must appear in the same order in which they occur in Ti.

*A particular sequence of the database operations (Read and Write) of a set of transactions is called a Schedule. The order of actions of every transaction in a schedule must always be the same as they were appearing in the transaction.*

In a DBMS, schedule is responsible for controlling the order in which transaction operations would be executed. The scheduler receives the transaction operations from the transaction manager. In order to control the execution sequence of transaction operations, the scheduler may delay a particular operation by placing it in a queue or may reject it.

### 8.3.1 Serial Schedules

A serial schedule is a schedule in which all the operations of one transaction are completed before any other transaction begins. It means there is no interleaving of the operations from multiple transactions.

Consider an example of two transactions T1 and T2. T1 transfers Rs. 1000 from Account A to Account C, while another transaction T2 transfers Rs. 500 from Account B to Account C. These transactions can be represented as:

| **Transaction T1** |
|---|
| Read A<br>A:= A-1000<br>Write A<br>Read C<br>C:=C+1000<br>Write (C) |

| **Transaction T2** |
|---|
| Read B<br>B:= B-500<br>Write (B)<br>Read C<br>C:=C+500<br>Write ( C) |

**Table 8.3 Transactions T1 and T2**

A serial Schedule for these two transactions will be as follows:

| **Transaction T1** | **Transaction T2** |
|---|---|
| Read A<br>A:= A-1000<br>Write A<br>Read C<br>C:=C+1000<br>Write (C) | |
| | Read B<br>B:= B-500<br>Write (B)<br>Read C<br>C:=C+500<br>Write (C ) |

**Table 8.4 Serial Schedule for Transactions T1 and T2**

This can also be represented in terms of the sequence of the operations from the two transactions T1 and T2 as follows:

Transaction T1: r1(A), w1(A), r1(C), w1 (C)
Transaction T2: r2(B), w2(B), r2 (C), w2 (C)

A Serial Schedule for these transactions will be:

T1 ⟶ T2,

which can also be represented by the sequence of its read and write operations as follows:

Schedule S: r1(A), w1(A), r1(C), w1 (C), r2 (B), w2 (B), r2 (C), w2 (C)

Another serial Schedule for these two transactions will be as follows:

| Transaction T1 | Transaction T2 |
|---|---|
| | Read B<br>B:= B-500<br>Write (B)<br>Read C<br>C:=C+500<br>Write (C ) |
| Read A<br>A:= A-1000<br>Write A<br>Read C<br>C:=C+1000<br>Write (C) | |

**Table 8.5 Another Serial Schedule for Transactions T1 and T2**

A Serial Schedule for these transactions will be:

T2 ⟶ T1,

which can also be represented by the sequence of its read and write operations as follows:

Schedule S: r2 (B), w2 (B), r2 (C), w2 (C), r1(A), w1(A), r1(C), w1 (C)

## 8.3.2 Non Serial Schedules and Serializability

Serial schedules are always correct, as transactions under serial schedules are independent to each other. Though, a transaction may not be consistent during the execution but on the completion of execution, it always results in consistent state.

However, serial schedule based approach is not practical and realistic, since one transaction may be waiting for some input/output from some secondary storage while the CPU remains idle, wasting a valuable resource. In some cases, a complex transaction if run serially may result in long wait for all other transactions waiting for it to finish.

Therefore it is required to prepare interleaved schedules which are not serial but produces the same results as were in the case of serial schedules. As serial schedules are consistent, therefore interleaved schedules must also produce the consistent solution.

According to this, the correctness of a concurrent transaction execution requires that such concurrent execution should provide the same result and have the same effect on the database as the one that is produced by the serial execution of the same transactions. A schedule that ensures this property is called a serializable schedule.

*A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions.*

Non-serial schedules may have several advantages over serial schedules including:

(a) Improved throughput and resource utilization: By executing several transactions at a time, throughput of the system will be increased as several operations in a computer system like I/O activities, CPU activities etc can run in parallel. Correspondingly, the processor and disk utilization also increases, thus enhancing the performance overall.

(b) Reduced waiting time: In serial execution, several transactions may have to wait for a preceding long transaction to complete, which way lead to an unwanted delay for executing important transactions. But in concurrent operations, several transactions are allowed to execute simultaneous, so small transaction having high priority may finish first, thus reduces the unpredictable delay.

The non-serial schedule for the transactions T1 and T2 is shown in Table 8.6 (a). This schedule produces the consistent solution.

| **Transaction T1** | **Transaction T2** |
|---|---|
| Read A<br>A:= A-1000<br>Write A | |
| | Read B<br>B:= B-500<br>Write (B) |
| Read C<br>C:=C+1000<br>Write (C) | |
| | Read C<br>C:=C+500<br>Write (C ) |

**Table 8.6(a) A Serializable Schedule**

One more example of non-serial (serializable) schedule that gives the same state of the database as in serial schedule is given:

| **T1** | **T2** |
|---|---|
| Read (A)<br>A=A-100<br>Write (A)<br>Read (B) | |
| | Read (A)<br>A=A*10<br>Write (A) |
| B = B + 100<br>Write (B) | |
| | Read (B)<br>B = A + B<br>Write (B) |

**Table 8.6(b) A Serializable Schedule**

However, not all non-serial (interleaved) schedules produce consistent solution. Consider the following schedule:

| Transaction T1 | Transaction T2 |
|---|---|
| Read A<br>A:= A-1000<br>Write A<br>Read C | |
| | Read B<br>B:= B-500<br>Write (B)<br>Read C<br>C:=C+500<br>Write (C ) |
| C:=C+1000<br>Write (C) | |

**Table 8.7 A Non-Serial (interleaved) Inconsistent Schedule**

It is therefore clear that many interleaved non-serial schedules would result in consistent state while many others will be in inconsistent states. All correct schedules of the two transactions T1 and T2 are serializable only if they are equivalent to either of these two serial schedules:

T1 ⟶ T2 or T2 ⟶ T1

## 8.3.3 Types of Serializability

As mentioned earlier, a non-serial schedule which is not equivalent to any serial schedule is not serializable. Two schedules are said to be equivalent if both the schedule produce the same final state of databases. The two main forms of serializability are:

1. Conflict Serializability
2. View Serializability

**Conflict Serializability:**

If two operations in a schedule satisfy the following three conditions then operations are said to conflict.

1. They belong to different transactions.
2. They access the same item
3. At least one of the operations is a write operation.

If schedule S can be transformed into a serial schedule S' by a series of swapping of non conflict operations, then S and S' are said to be *conflict equivalent*, which leads to conflict serializability. We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

**View Serializability**

Consider to schedules S1 and S2 where same set of instructions participate in both schedules. The schedules S1 and S2 are said to be *view equivalent* if the following conditions are met.

1. For each data item A, if transaction Ti reads the initial value of A in schedule S1, the transaction Ti must, in schedule S2, also read the initial value of A.

2. For each data item A, if transaction Ti executes Read (A) in schedule S1, and the value was produced by transaction Tj (if any), then transaction Ti must in schedule S2 also read the value of A that was produced by transaction Tj.

3. For each data item A, the transaction (if any) that performs the final Write (A) operation in schedule S1 must perform the final Write (A) operation in schedule S2.

A schedule S is view serializable it is view equivalent to a serial schedule. Every conflict serializable schedule is also view serializable.

For example, the following two schedules S1 and S2 are not conflict equivalent, but they are view equivalent.

| T1 | T2 | T3 |
|---|---|---|
| Read (A) | | |
| | Write (A) | |
| Write (A) | | |
| | | Write (A) |

| T1 | T2 | T3 |
|---|---|---|
| Read (A) | | |
| Write (A) | | |
| | Write (A) | |
| | | Write (A) |

**Table 8.8 View Equivalent Schedules S1 and S2**

Here, it is also to note that Schedule S1 is not conflict serializable, but it is view serializable.

## 8.3.4 Testing for Serializability

To test a given schedule S for conflict serializability, we construct a directed graph, which is known as precedence graph. Then we invoke a cycle detection algorithm. If it detects a cycle then schedule is known as non-conflicting serializable schedule.

The graph consists of a pair of G = (V,E), where V is the set of vertices and E is the set of edges. The set of vertices consists of all the transactions executing in the schedule, whereas the set of edges consists of all the edges Ti →Tj for which one of the following three conditions hold:

1. Ti executes Write (A) before Tj executes Write (A)
2. Ti executes Read (A) before Tj executes Write (A)
3. Ti executes Write (A) before Tj executes Write (A)

If an edge Ti→Tj exists in the precedence graph, it means that in any serial schedule S' equivalent to S, Ti must appear before Tj. Once again consider the same schedule given in Table 8.3:

| Transaction T1 | Transaction T2 |
|---|---|
| Read A<br>A:= A-1000<br>Write A<br>Read C<br>C:=C+1000<br>Write (C) | |
| | Read B<br>B:= B-500<br>Write (B)<br>Read C<br>C:=C+500<br>Write (C ) |

**Table 8.4 Serial Schedule for Transaction T1 and T2**

For this schedule, the precedence graph will be as:

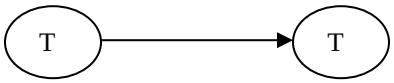


**Fig. 8.3 Precedence Graph for Schedule of Table 8.3**

It contains the single edge T1→T2, since all the operations of T1 are executed before the first operation of T2 is executed.

Now consider another schedule:

| Transaction T1 | Transaction T2 |
|---|---|
| Read A<br>A:= A-1000<br>Write A | |
| | Read A<br>A:= A+500<br>Write (A) |
| Read C<br>C:=C+1000<br>Write (C) | |
| | Read C<br>C:=C-500<br>Write (C ) |

**Table 8.9 Serializable Schedule**

For this schedule, the precedence graph will be as:

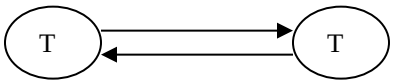

**Fig. 8.4 Precedence Graph for Schedule of Table 8.3**

It contains the edge T1→T2, because T1 executes Read (A) before T2 executes Write (A). Also, it contains T2→T1 because T2 executes Read (C) before T1 executes write (Y).

So, if precedence graph for S has a cycle then schedule S is not Conflict Serializable, but if graph does not contain any cycle, then schedule is a conflict serializable. Every conflict serializable schedule is view serializable, but there are view serializable schedules which are not conflict serializable.

## 8.5 RECOVERABLE SCHEDULES

A schedule S is recoverable if:

- No transaction T1 in S commits until all transactions T2, that have written an item that T reads, have committed.
- A transaction T1 reads from transaction T2 in a schedule S if some item X is first written by T2 and read later by T1.
- In addition, T2 should not be aborted before T1 reads item X, and there should be no transactions that write X after T2 writes it and before T1 reads it (unless those transactions, if any, have aborted before T1 reads X)

**Example:** Consider the Schedule S having two transactions T1 and T2 with the following operations:

S: r1(X); r2(X); w1(X); r1(Y); w2(X); c2; w1(Y); c1;

Two commit operations have been issued. Is S recoverable?

Yes, even though it suffers from the lost update problem.

If the transactions T1 and T2 are submitted at about the same time, and suppose that their operations are interleaved. If T2 reads a value of X before T1 changes it in the database, the update from T1 is lost.

**Example:** Consider another example

S: r1(X); w1(X); r2(X); r1(Y); w2(X); c2; a1;

S is not recoverable, because T2 reads item X from T1, and then T2 commits before T1 commits. If T1 aborts after the c2 operation in S, then the value of X that T2 read is no longer valid and T2 must be aborted after it had been committed. *This schedule is not recoverable.*

For the schedule to be recoverable, the c2 operation in S must be postponed until after T1 commits, else both must abort because the X that T2 reads is no longer valid.

## Exercise

1. Explain the distinction between the terms serial schedule and serialization schedule.
2. What benefit is provided by strict two phase locking?
3. Explain how concurrency can lead to inconsistency. What is deadlock? Can it occur in a serializable schedule? If so, give an example? How can it be detected and resolved?

4.1 Which one of the following is not a proper state of transaction?

a) Partially aborted
b) Partially committed
c) Aborted
d) Committed

4.2 Which of the following concurrency control schemes is not based on the serializability property?

a) Two phase locking
b) Graph based locking
c) Time based locking
d) None of these

4.3 Rollback of transactions is normally used to recover from transaction failure

a) update the transaction
b) retrieve old records
c) repeat a transaction
d) None of these

4.4 Which of the following describes the time-stamp based protocols correctly?

(a) This protocol requires that each transaction issue lock and unlock requests in two phases.
(b) This protocol employs only exclusive locks.
(c) This protocol selects an ordering among transactions in advance.
(d) None of the above

4.5 Transactions are initiated by BEGIN TRANSACTION and terminated

a) by COMMIT TRANSACTION
b) by ROLL BACK TRANSACTION
c) either by COMMIT TRANSACTION or by ROLLBACK TRANSACTION
d) None of these