Transactions

Notes by Mannan Ul Haq

When the data of users is stored in a database, that data needs to be accessed and modified from time to time. This task should be performed with a specified set of rules and in a systematic way to maintain the consistency and integrity of the data present in a database. In DBMS, this task is called a transaction.

Operations of Transaction

Transaction in Database Management Systems (DBMS) can be defined as a set of logically related operations.

A user can make different types of requests to access and modify the contents of a database. So, we have different types of operations relating to a transaction.

i) Read(X)

A read operation is used to read the value of X from the database and store it in a buffer in the main memory for further actions such as displaying that value.

ii) Write(X)

A write operation is used to write the value to the database from the buffer in the main memory. For a write operation to be performed, first a read operation is performed to bring its value in buffer, and then some changes are made to it, then to store the modified value back in the database, a write operation is performed.

iii) Commit

This operation in transactions is used to maintain integrity in the database. Due to some failure of power, hardware, or software, etc., a transaction might get interrupted before all its operations are completed. This may cause ambiguity in the database, i.e. it might get inconsistent before and after the transaction. To ensure that further operations of any other transaction are performed only after work of the current transaction is done, a commit operation is performed to the changes made by a transaction permanently to the database.

iv) Rollback

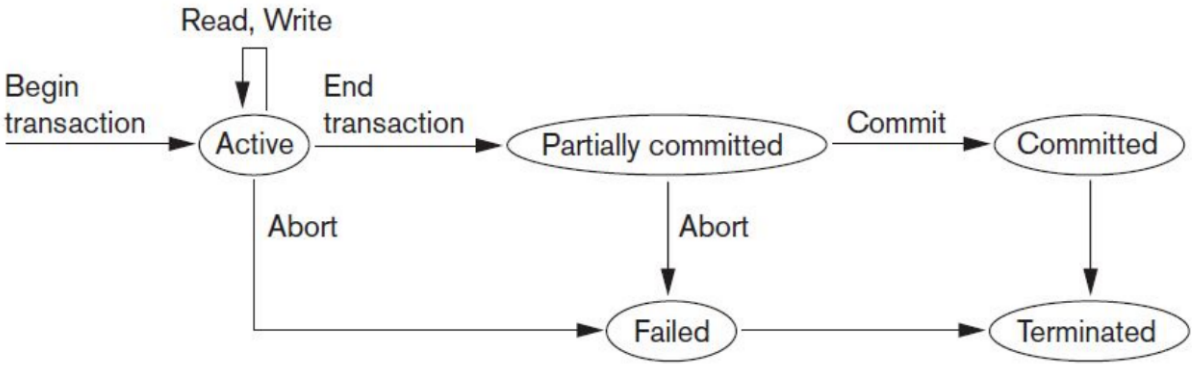
This operation is performed to bring the database to the last saved state when any transaction is interrupted in between due to any power, hardware, or software failure. In simple words, it can be said that a rollback operation does undo the operations of transactions that were performed before its interruption to achieve a safe state of the database and avoid any kind of ambiguity or inconsistency.

Properties of Transactions

Transactions in a database ensure consistency and integrity through four key properties, known as ACID properties:

1. **Atomicity**
    - **All or Nothing**: A transaction must either complete fully or not at all. Partial completion is not allowed.
    - **Commit and Rollback**: Changes are saved only if the transaction is successful. If interrupted, all changes are undone to revert the database to its previous state.
2. **Consistency**
    - **State Integrity**: The database must remain consistent before and after the transaction.
    - **Logical Operations**: Only valid data changes as intended by the user are made, ensuring no ambiguity.
3. **Isolation**
    - **Concurrent Transactions**: Transactions must not interfere with each other.
    - **Data Access**: Data being used by one transaction cannot be accessed by another until the first transaction is complete.
    - **Concurrency Control**: Managed by the DBMS to prevent conflicting transactions.
4. **Durability**
    - **Permanent Changes**: Once a transaction is complete, changes are permanent, even in case of a system failure.
    - **Recovery**: The DBMS ensures that the database remains in a consistent state after a crash or failure.

Transaction States



Concurrency problems in Transactions

Concurrency control is an essential aspect of database management systems (DBMS) that ensures transactions can execute concurrently without interfering with each other. However, concurrency control can be challenging to implement, and without it, several problems can arise, affecting the consistency of the database.

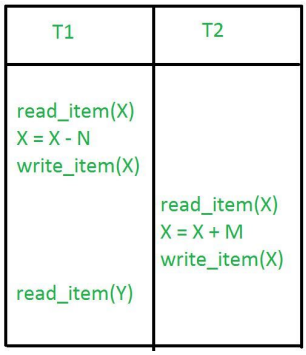
When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems. These problems are commonly referred to as concurrency problems in a database environment.

The five concurrency problems that can occur in the database are:

1. Temporary Update Problem:

Temporary update or dirty read problem occurs when one transaction updates an item and fails. But the updated item is used by another transaction before the item is changed or reverted back to its last value.

Example:

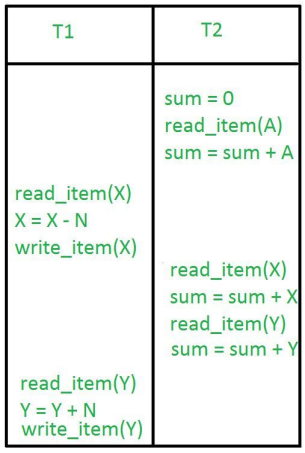


In the above example, if transaction 1 fails for some reason then X will revert back to its previous value. But transaction 2 has already read the incorrect value of X.

2. Incorrect Summary Problem:

Consider a situation, where one transaction is applying the aggregate function on some records while another transaction is updating these records. The aggregate function may calculate some values before the values have been updated and others after they are updated.

Example:

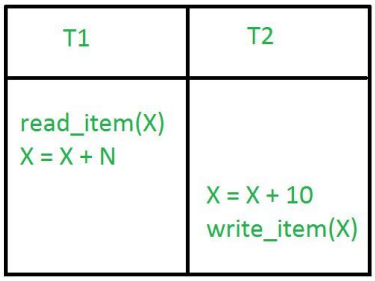


In the above example, transaction 2 is calculating the sum of some records while transaction 1 is updating them. Therefore the aggregate function may calculate some values before they have been updated and others after they have been updated.

3. Lost Update Problem:

In the lost update problem, an update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction.

Example:

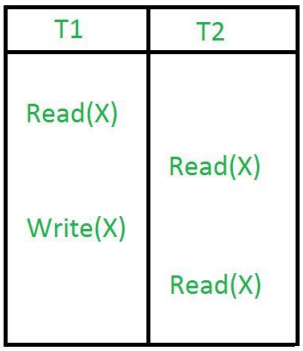


In the above example, transaction 2 changes the value of X but it will get overwritten by the write commit by transaction 1 on X *(not shown in the image above)*. Therefore, the update done by transaction 2 will be lost. Basically, the write commit done by the **last transaction** will overwrite all previous write commits.

4. Unrepeatable Read Problem:

The unrepeatable problem occurs when two or more read operations of the same transaction read different values of the same variable.

Example:

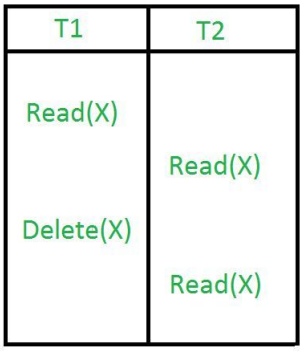


In the above example, once transaction 2 reads the variable X, a write operation in transaction 1 changes the value of the variable X. Thus, when another read operation is performed by transaction 2, it reads the new value of X which was updated by transaction 1.

5. Phantom Read Problem:

The phantom read problem occurs when a transaction reads a variable once but when it tries to read that same variable again, an error occurs saying that the variable does not exist.

Example:



In the above example, once transaction 2 reads the variable X, transaction 1 deletes the variable X without transaction 2's knowledge. Thus, when transaction 2 tries to read X, it is not able to do it.

Schedule

Schedule is process of grouping transactions into one and executing them in a predefined order. Schedule is required in database because when multiple transactions execute in parallel, they may affect the result of each other. So, to resolve this the order of the transactions are changed by creating a schedule.

Types of Schedules:

1. **Serial Schedule:** A schedule in which the transactions are defined to execute one after another is called serial schedule.
2. **Non- Serial Schedule:** A schedule in which the transactions are defined to execute in any order is called non-serial schedule.

Recoverable and Non-Recoverable Schedule:

- Schedules in which transactions commit only after all transactions whose changes they commit are called recoverable schedule.
- If some transaction T(j) is reading value updated or written by some other transaction T(i) then, the commit of T(j) must occur after the commit of T(i).

Schedule S1		**Schedule S2**	
T1	**T2**	**T1**	**T2**
R(X) X=X+10 W(X)		R(X) X=X+10 W(X) COMMIT	

Rollback COMMIT	R(X) X=X-5 W(X) COMMIT		R(X) X=X-5 W(X) COMMIT
Non-Recoverable Schedule		**Recoverable Schedule**	

Cascading Abort and Cascadeless Schedule:

- If in a schedule, a transaction is not allowed to read a data item until the last transaction that has written it is committed or aborted, then such a schedule is called Cascadeless schedule.

Schedule S1			**Schedule S2**		
T1	**T2**	**T3**	**T1**	**T2**	**T3**
(Cascading Rollback) R(X) W(X)			R(X) W(X) COMMIT		
	(Cascading Rollback) R(X) W(X)			R(X) W(X) COMMIT	
If transaction fails→ COMMIT		(Cascading Rollback) R(X) W(X) COMMIT			R(X) W(X) COMMIT
Cascading Abort			**Cascadeless Schedule**		

Strict Schedule:

- If in a schedule, a transaction is neither allowed to read nor write a data item until the last transaction that has written it is committed or aborted, then such schedule is called strict schedule.
- Strict schedule implements more restrictions than cascadeless schedule.

Schedule S	
T1	**T2**
R(X) W(X) COMMIT	
	W(X) R(X) COMMIT
Strict Schedule	

Levels of Isolation in DBMS

1. **Level 0 Isolation**
    - **Dirty Reads Allowed**: Can read uncommitted changes.
    - **No Overwrites of Higher-Level Dirty Reads**.
2. **Level 1 Isolation (Read Uncommitted)**
    - **No Lost Updates**: Prevents lost updates.
    - **Dirty Reads Allowed**.
3. **Level 2 Isolation (Read Committed)**
    - **No Lost Updates**.
    - **No Dirty Reads**: Only reads committed changes.
4. **Level 3 Isolation (Repeatable Read)**
    - **No Lost Updates**.
    - **No Dirty Reads**.
    - **Repeatable Reads**: Ensures consistent data across multiple reads.
5. **Snapshot Isolation**
    - **No Lost Updates**.
    - **No Dirty Reads**.
    - **Consistent Snapshot**: Each transaction sees a consistent database state.
    - Ensures phantom record problem will not occur

	Type of Violation		
Isolation Level	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Serializable Transactions

Serializable is the highest level of isolation in DBMS, ensuring that the results of executing transactions concurrently are the same as if they were executed sequentially, one after the other.

Conflict Serializable

A schedule (sequence of operations from multiple transactions) is conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

Checking Conflict Serializable

To determine if a schedule is conflict serializable, follow these steps:

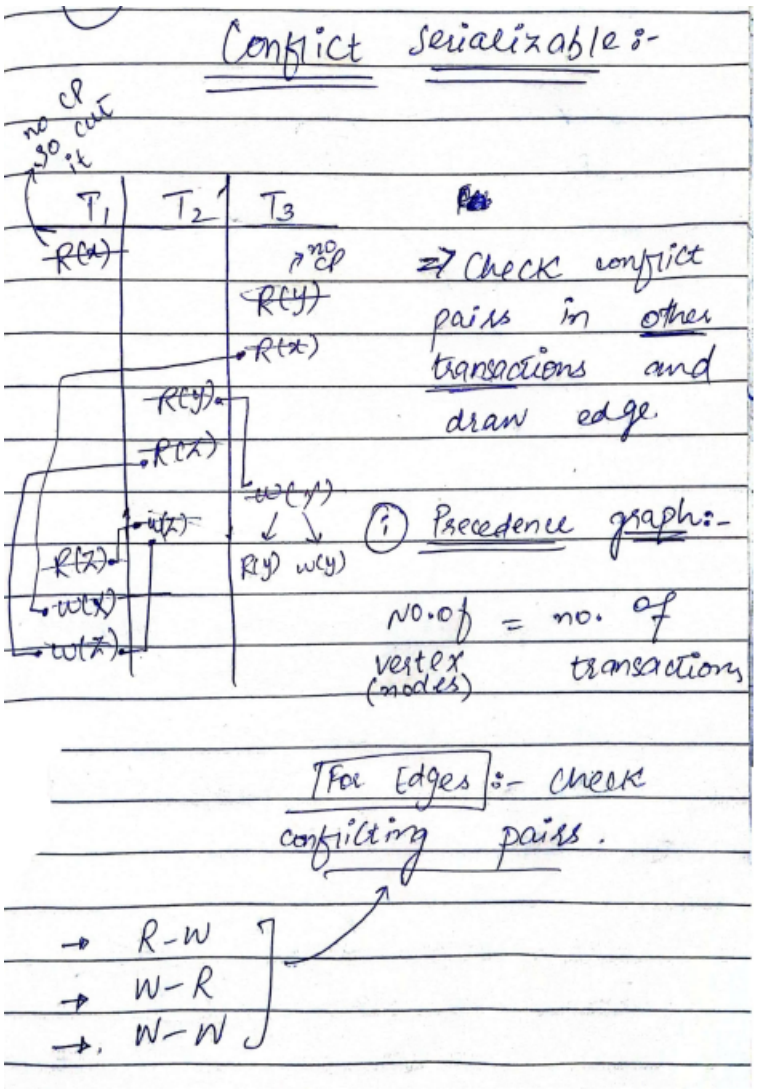
1. **Identify Conflicts**: Find all pairs of conflicting operations.

   Conflicting Operations: Two operations are in conflict if:

   - They belong to different transactions.
   - They access the same data item.
   - At least one of them is a write operation.

2. **Construct a Precedence Graph**:
   - **Nodes**: Each node represents a transaction.
   - **Edges**: Draw a directed edge from transaction *Ti* to transaction *Tj* if an operation in *Ti* precedes and conflicts with an operation in *Tj*.
3. **Check for Cycles**:
   - If the precedence graph has no cycles, the schedule is conflict serializable.
   - If there are cycles, the schedule is not conflict serializable.

Conflict serializable :-



⇒ Check conflict pairs in other transactions and draw edge.

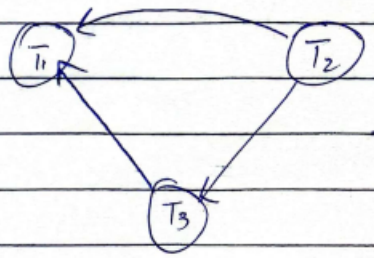
(i) Precedence graph :-

No. of vertex (nodes) = no. of transactions

For Edges :- check conflicting pairs.

- R-W
- W-R
- W-W

⇒ Precedence graph:-



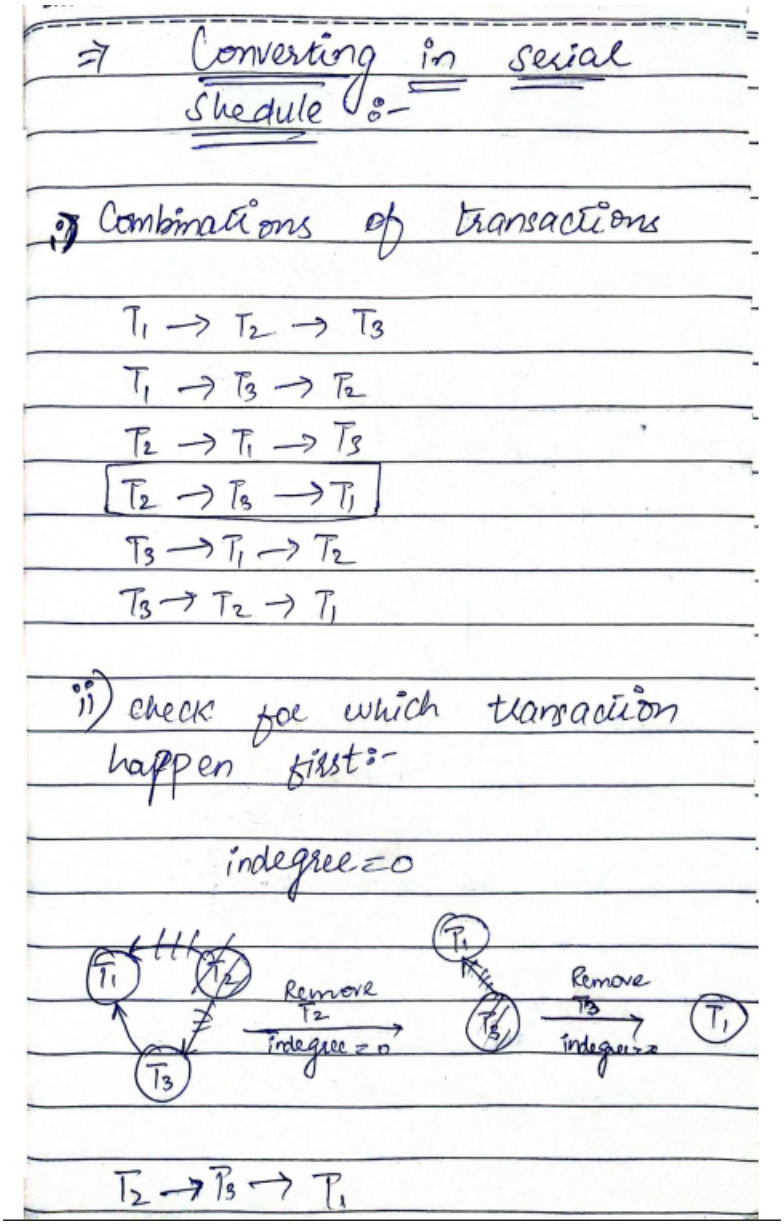
→ no loop

⇒ Check for loop / cycle in graph.

⇒ if there is no loop or cycle then it is conflict serializable

⇒ if shedule is CS then it is serializable means serial shedule exist in equivalence.

⇒ if graph is Serializable then it is consistant.



View Serializable

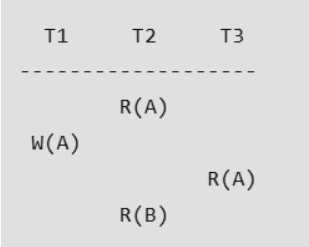
View serializability is a concept used to ensure that concurrent execution of transactions in a schedule produces the same results as some serial execution of the transactions. This is more relaxed compared to conflict serializability. Here's a breakdown of the concepts:

Checking View Equivalence

Two schedules are **view-equivalent** if they produce the same results when executed on the same initial database state. For two schedules S1 and S2 to be view-equivalent, they must satisfy three conditions:

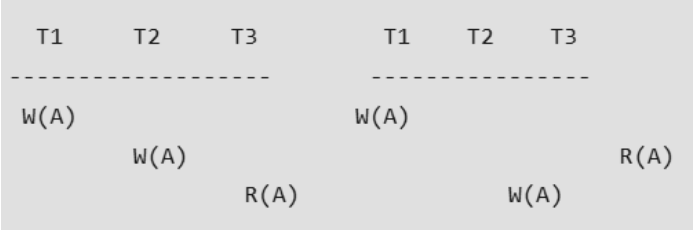
1. **Initial Read**:

If a transaction T1 reads a data item A from the database in S1, then in S2 also T1 should read A from the same initial value.



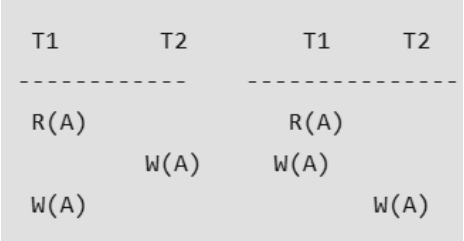
2. **Updated Read**:

   If a transaction T_i reads a data item A that was updated by T_j in S1, then in S2 also T_i should read A which was updated by T_j.



3. **Final Write**:

   If a transaction T1 performs the final write on data item A in S1, then T1 should perform the final write on A in S2 as well.



Example:

Non-Serial

S1

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Serial

S2

T1	T2
R(A)	
W(A)	
R(A)	
W(A)	
	R(B)
	W(B)
	R(B)
	W(B)

Let's look at the three view serializability conditions:

1. **Initial Read**

   First, we will see that transaction T1 in schedule S1 reads data item A first. Transaction T1 in S2 also reads data item A first. Then look for B. Transaction T1 in schedule S1 reads data item B first. T1 also performs the first read operation on B in S2. We checked both data items A and B, and the initial read condition is met in S1 and S2.

2. **Update Read**

   The second method is about update read operation. Transaction T2 in S1 reads the value of A written by T1. In the same transaction, T2 reads the A after it is written by T1 in S2. Transaction T2 in S1 reads the value of B written by T1. The same transaction T2 reads the value of B after it has been updated by T1 in S2. Both schedules also satisfy the update read condition.

3. **Final Write**

   In schedule S1, the final write operation on A is done by transaction T2. Transaction T2 also performs the final write on A in S2. Let's look for B. Transaction T2 performs the final write operation on B in schedule S1. T2 completes the final write on B in schedule S2. We checked both data items A and B, and the final write condition is met in S1 and S2.