

Module 5

Single-User versus Multiuser Systems

- At most one user can use the system at a time is called as single user system
- Many users can access the system concurrently is called as multi user system
- The term concurrency is used in Multi user system
- Concurrent execution of processes is interleaved in a single CPU

TRANSACTION

- It is **a group of database commands that can change/ access the data stored in a database**
 - A transaction is treated as a **single unit of work**, either all statements are succeeded or none of them
 - **Transaction maintains integrity of data in a database**
 - Eg, Transfer Rs. 100 from A to B:
 - **Transaction boundaries:** Begin and End transaction.

Simple Transaction Example

1. Read A's account balance
2. Deduct the amount from A
3. Write the remaining balance
4. Read B's account balance
5. Add the amount to B's account balance
6. Write the new updated balance

- This whole set of operations can be called a transaction.
- The transaction can have operations like read, write, insert, update, delete.
- In DBMS, we write the above 6 steps transaction like this:

BEGIN TRANSACTION

R(A);

$A = A - 10000;$

W(A);

R(B);

$B = B + 10000;$

W(B);

END TRANSACTION

In the above transaction **R** refers to the Read operation and **W** refers to the write operation.

Transaction failure in between the operations

- The main problem that can happen during a transaction is that the **transaction can fail before finishing the all the operations in the set**. This can happen due to power failure, system crash etc.
- This is a serious problem that can **leave database in an inconsistent state**.
- Assume that transaction fail after third operation (see the example above) then the amount would be deducted from your account but your friend will not receive it.
- **To solve this problem**, we have the following **two operations** :
 - **Commit** : If all the operations in a transaction are completed successfully then commit those changes to the database permanently.
 - **Rollback** : If any of the operation fails then rollback all the changes done by previous operations
- Even though these operations can help us avoiding several issues that may arise during transaction but they are not sufficient when two transactions are running concurrently

TERMINOLOGY

- BEGIN TRANSACTION
 - Beginning of transaction execution
- END TRANSACTION
 - Transaction execution is completed
- ROLLBACK_TRANSACTION (ABORT)
 - This signals that the transaction has ended unsuccessfully, so that any changes made by that transaction must be undone
- COMMIT_TRANSACTION
 - Signals successful end of the transaction. All changes made by the transaction are recorded permanently in the database and will not be undone
- A single **application program** may contain several transactions separated by the Begin and End transaction boundaries.
- If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a read-only transaction; otherwise it is known as a **read- write transaction**.

- **A database** is a collection of named data items
- **Granularity** of data
 - the size of a data item
 - a field, a record , or a whole disk block (Transaction processing Concepts are independent of granularity)
- Basic database access operations are **read** and **write**
 - **read_item(X)**: Reads a database item named X into a program variable also named X.
 - **write_item(X)**: Writes the value of program variable X into the database item named X.
- Basic unit of data transfer from the disk to the computer main memory is one block.
- A data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.

read_item(X) command

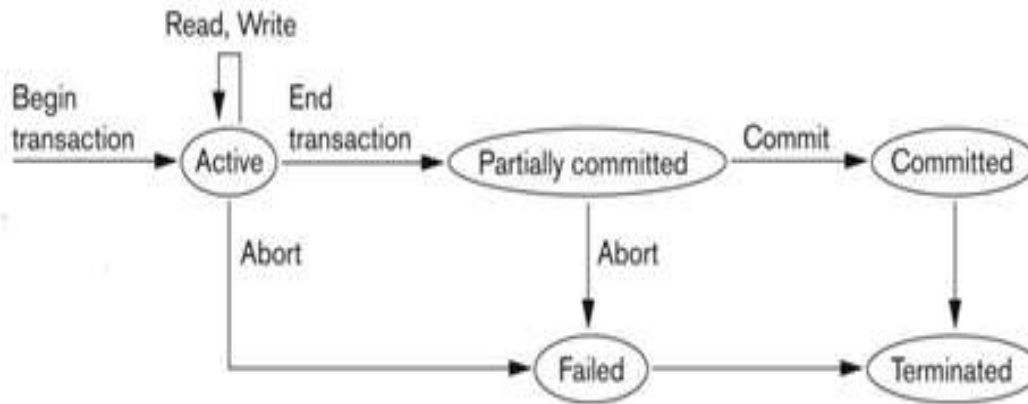
- **Find** the address of the disk block that contains itemX.
- **Copy** that disk block into a buffer in main memory
- Copy item X from the buffer to the program variable named X.

write_item(X) command

- Find the address of the disk block that contains item X.
- Copy that disk block into a buffer in main memory
- Copy item X from the program variable named X into its correct location in the buffer.
- Store the updated block from the buffer back to disk

i.e. **it writes the value of program variable X into the database item named X**

STATES OF A TRANSACTION / TRANSACTION STATES



- **Active State**

Statements inside the transaction block are executed

- **Partially Committed**

- A transaction goes into “partially committed” state from the active state when there are read and write operations present in the transaction.
- A transaction contains number of read and write operations.
- Once the whole transaction is successfully executed, the transaction goes into partially committed state where we have all the read and write operations performed on the main memory (local memory) instead of the actual database.

- This state helps us to rollback the changes made to the database in case of a failure during execution

All updates/changes made by the transaction is applied on the data block available in the Main Memory Buffer. It is not yet updated in actual data block(Secondary Storage)

- **Committed State**

- If a transaction completes the execution successfully then all the changes made in the local memory during partially committed state are permanently stored in the database.
- A transaction goes from partially committed state to committed state when everything is successful.

Whenever the changes made by the transaction are permanently recorded in secondary storage , then we can say that the transaction is committed state. Now the changes cant be undone

- **Aborted State or Failed State**

- If a transaction fails during execution then the transaction goes into a failed state.
- The changes made into the local memory (or buffer) are rolled back to the previous consistent state and the transaction goes into aborted state from the failed state.

- **Terminated State**

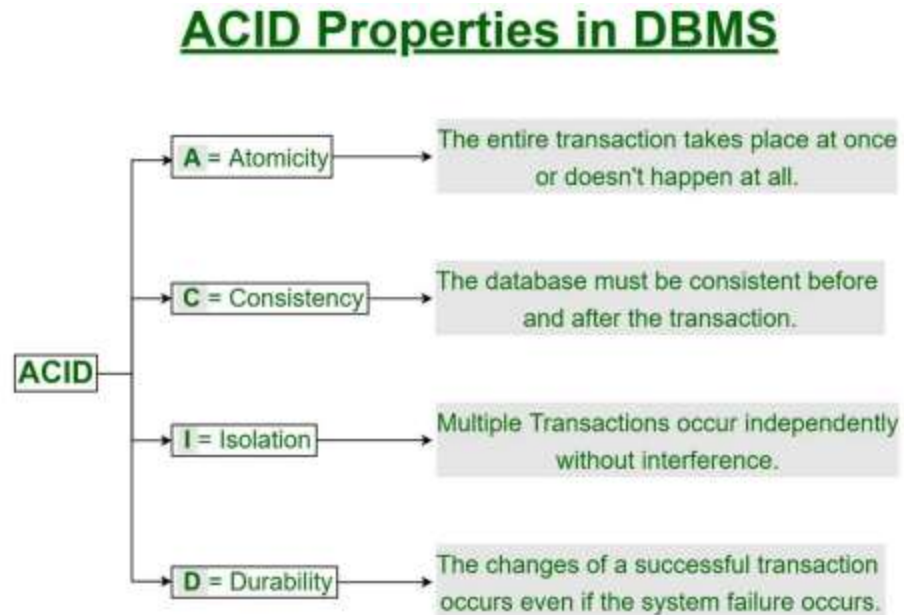
- Corresponds to transaction leaving the system

Desirable Properties of a Transaction

ACID properties:

- A transaction is a single logical unit of work which accesses and possibly modifies the contents of a database.
- Transactions access data using read and write operations.
- In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called ACID properties.

1. **Atomicity**
2. **Consistency**
3. **Isolation**
4. **Durability**



1. Atomicity

- A transaction is said to be atomic **if either all of the commands are succeeded or none of them**
- i.e. transactions do not occur partially.
- It involves the following two operations.
 - Abort : If a transaction aborts, changes made to database are not visible.
 - Commit : If a transaction commits, changes made are visible.
- Atomicity is also known as the '**All or nothing rule**'.
- **Responsibility of Recovery Subsystem**
- Eg: Transfer Rs. 100 from A to B:

BEGIN TRANSACTION

R(A);

A=A-100;

W(A);

R(B);

B=B+100;

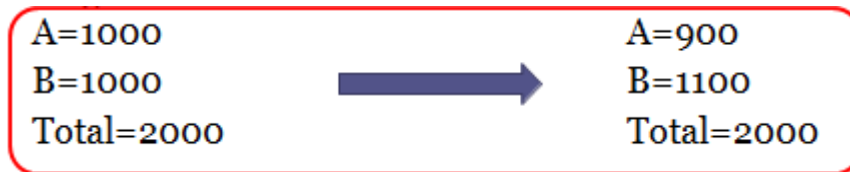
W(B);

END TRANSACTION

Either execute all command or no
command at all

2. Consistency

- This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. (It should take the database from one consistent state to another)
- ie, **A transaction should be consistency preserving**
- It refers to the correctness of a database.
- Responsibility of Programmers who wrote the database programs.
- Referring to the example The total amount before and after the transaction must be maintained.
- Eg: Transfer Rs. 100 from A to B



3. Isolation

- This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state.
- Transactions occur independently without interference.
- Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed.
- This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order
- In simple , **Transactions should be isolate to each other during concurrent execution**
- Responsibility: Concurrent control subsystem

4. Durability or Permanency

- This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs.
- These updates now become permanent and are stored in non-volatile memory.
- The effects of the transaction, thus, are never lost.
- Responsibility of Recovery Subsystem

The **ACID properties**, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results

Why Concurrency Control is needed ? or Concurrency Control Issues

- Concurrency Control in DBMS is a procedure of **managing simultaneous transactions** ensuring their atomicity, isolation, consistency and serializability.
- In a database transaction, the two main operations are **READ** and **WRITE** operations.
- So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the **data may become inconsistent**.
- So, the following problems occur with the Concurrent Execution of the operations:
 - Lost Update Problem
 - Temporary Update Problem
 - Incorrect Summary Problem
 - Unrepeatable Read Problem
 - Phantom Read Problem

Lost Update : nullifies the update of the first transaction violating ACID properties

Dirty Read : reading uncommitted data

Unrepeatable Read Problem : different value for same data set variable

Incorrect Summary Problem : invalid result on aggregating data

Phantom Read : losing the variable or data set on second operation

Problem 1: **Lost Update Problems (W - W Conflict)**


- A lost update problem occurs due to the update of the same record by two different transactions at the same time.
- In simple words, when two transactions are updating the same record at the same time in a DBMS then a lost update problem occurs.
- The first transaction updates a record and the second transaction updates the same record again, which nullifies the update of the first transaction.
- As the update by the first transaction is lost this concurrency problem is known as the lost update problem.

When two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent. This concurrency problem is known as the lost update problem.

- We need to control and prevent lost update concurrency problems.
- There are numerous ways **to prevent lost update problem** and ensure that update from any transaction is not lost.
 - Optimistic Locking
 - Atomic Write Operations

Eg :

Time



		X	
Transaction A	Transaction B	A	B
Read (X) $X = X + 15$		$X = 100$ 115	
	Read (X) $x = x - 25$ WRITE (X)		$X = 100$ $X = 75$ $X = 75$
WRITE (X)		$X = 115$	

- At time t1, transaction T_X reads the value of 100
- At time t2, transaction T_X adds 15 to X (only adds and not updated/write).
- Alternately, at time t3, transaction T_X reads the value of 100 (because the value of X is not updated yet)
- At time t4, transaction T_X deducts 25 from X , that becomes 75 (only deducted but not updated/write).
- At time t6, transaction T_X writes the value X that will be updated as 75
- At time t7, transaction T_X writes the values as 115.ie, It means the value written by T_X is lost, i.e., 75 is lost.

Problem 2 : **Dirty Read Problems (W-R Conflict)**

- It is also called as **Temporary Update Problem**
- This problem occurs when a transaction reads the data that has been updated by another transaction that is still uncommitted.
- It arises due to multiple uncommitted transactions executing simultaneously.

The dirty read problem occurs when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.

Eg :

- Consider two transactions A and B performing read/write operations on a data DT in the database DB.
- The current value of DT is 1000:
- The following table shows the read/write operations in A and B transactions.

Time	A	B
T1	READ(DT)	-----
T2	DT=DT+500	-----
T3	WRITE(DT)	-----
T4	-----	READ(DT)
T5	-----	COMMIT
T6	ROLLBACK	-----

- Transaction A reads the value of data DT as 1000 and modifies it to 1500 which gets stored in the temporary buffer.
- The transaction B reads the data DT as 1500 and commits it and the value of DT permanently gets changed to 1500 in the database DB.
- Then some server errors occur in transaction A and it wants to get rollback to its initial value, i.e., 1000 and then the dirty read problem occurs.

Problem 3 : Unrepeatable Read Problem (W-R Conflict)

- A transaction T reads the same item twice and the item is changed by another transaction T' between the two reads. Hence, T receives different values for its two reads of the same item, this may occur

The unrepeatable read problem occurs when two or more different values of the same data are read during the read operations in the same transaction.

Eg: Consider two transactions A and B performing read/write operations on a data DT in the database DB. The current value of DT is 1000:

Time	A	B
T1	READ(DT)	-----
T2	-----	READ(DT)
T3	DT=DT+500	-----
T4	WRITE(DT)	-----
T5	-----	READ(DT)

Transaction A and B initially read the value of DT as 1000. Transaction A modifies the value of DT from 1000 to 1500 and then again transaction B reads the value and finds it to be 1500. Transaction B finds two different values of DT in its two different read operations.

Problem 4 : **Incorrect Summary Problem**

- The Incorrect summary problem **occurs when there is an incorrect sum of the two data.**
- This happens when a transaction tries to sum two data using an aggregate function and the value of any one of the data get changed by another transaction.

Eg: Consider two transactions A and B performing read/write operations on two data DT1 and DT2 in the database DB. The current value of DT1 is 1000 and DT2 is 2000

Time	A	B
T1	READ(DT1)	-----
T2	add=0	-----
T3	add=add+DT1	-----
T4	-----	READ(DT2)
T5	-----	DT2=DT2+500
T6	READ(DT2)	-----
T7	add=add+DT2	-----

- Transaction A reads the value of DT1 as 1000.
- It uses an aggregate function SUM which calculates the sum of two data DT1 and DT2 in variable add but in between the value of DT2 get changed from 2000 to 2500 by transaction B.
- Variable add uses the modified value of DT2 and gives the resultant sum as 3500 instead of 3000.

Problem 5 : Phantom Read Problem

- In the phantom read problem, data is read through two different read operations in the same transaction.
- In the first read operation, a value of the data is obtained but in the second operation, an error is obtained saying the data does not exist.

Eg: Consider two transactions A and B performing read/write operations on a data DT in the database DB. The current value of DT is 1000

Time	A	B
T1	READ(DT)
T2	READ(DT)
T3	DELETE(DT)
T4	READ(DT)

- Transaction B initially reads the value of DT as 1000.
- Transaction A deletes the data DT from the database DB and then again transaction B reads the value and finds an error saying the data DT does not exist in the database DB.

Why recovery is needed?

- **What causes a Transaction to fail ?**

- 1. A computer failure (system crash)**

- A hardware or software error occurs in the computer system during transaction execution.
- If the hardware crashes, the contents of the computer's internal memory may be lost.

- 2. A transaction or system error**

- Transaction failure may also occur because of erroneous parameter values or because of a logical programming error.
- In addition, the user may interrupt the transaction during its execution.

- 3. Local errors or exception conditions detected by the transaction**

- Certain conditions necessitate cancellation of the transaction.
- A programmed abort in the transaction causes it to fail.

- 4. Concurrency control enforcement**

- The concurrency control method may decide to abort the transaction, because it violates serializability

5. Disk failure

- Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash.
- This may happen during a read or a write operation of the transaction.

6. Physical problems and catastrophes

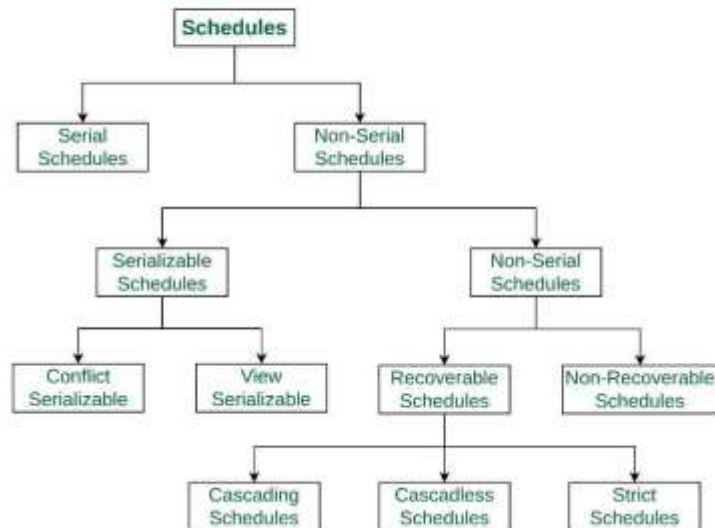
- This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, overwriting disks etc

SCHEDULES OF TRANSACTIONS

- A series of operation from one transaction to another transaction is known as schedule (or history)
- When there are multiple transactions that are running in a concurrent manner and the order of operation is needed to be set so that the operations do not overlap each other
- Ie, It is used to preserve the order of the operation in each of the individual transaction.

When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as a transaction schedule (or history).

Types of schedules in DBMS



1. Serial Schedules

- Schedules in which the transactions are executed non-interleaved. i.e., a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules.
- In Serial schedule, a transaction is executed completely before starting the execution of another transaction.
- In other words, in serial schedule, a transaction does not start execution until the currently running transaction finished execution.
- In simple , **Entire transactions are performed in serial order and Only One transaction is active at a time**
- This type of execution of transaction is also known as **non interleaved execution**.
- Eg : Consider the following schedule involving two transactions T1 and T2.

T ₁	T ₂
R(A)	
W(A)	
R(B)	
	W(B)
	R(A)
	R(B)

- where R(A) denotes that a read operation is performed on some data item 'A'
- This is a serial schedule since the transactions perform serially in the order T1 → T2

- Every serial schedule is considered as correct if the transactions are independent
- it does not matter which transaction is executed first
- $T_1 T_2 T_3 \dots T_n = T_2 T_3 T_4 T_1 \dots T_n = T_3 T_n T_1 T_4 T_6 \dots T_{10}$
- All these serial schedules are correct schedules

Problems with serial schedules

1. Poor resource utilization
2. More waiting time
3. Less throughput
4. Late response
5. Reduced Concurrency
6. Decreased System Performance
7. Increased Overhead

Serial schedules are unacceptable in practice even though they are correct

2. Non-Serial Schedule

- This is a type of Scheduling where the operations of **multiple transactions are interleaved**.
- This might lead to arise in the concurrency problem.
- The transactions are executed in a non serial manner, keeping the end result correct and same as the serial schedule.
- Unlike the serial schedule where one transaction must wait for another to complete all its operation, in the non-serial schedule, the other transaction proceeds without waiting for the previous transaction to complete.
- This sort of schedule does not provide any benefit of the concurrent transaction.
- The **Non-Serial Schedule can be divided further into Serializable and Non-Serializable**.

Eg : **Schedule 1**

T1
R(A)
A=A+50
W(A)

R(B)
B=B-30
W(B)

T2

R(A)
A=A+40
W(A)

R(B)
B=B-60
W(B)

Initially	Final Value
A=100	A=190
B=200	B=110

Schedule S1 gives the correct result same as serial schedule

Schedule 2

T1
R(A)
A=A+50

W(A)
R(B)
B=B-30

W(B)

T2

R(A)
A=A+40
W(A)

R(B)
B=B-60
W(B)

Initially	Final Value
A=100	A=150
B=200	B=170

Schedule S2 gives the erroneous result not same as serial schedule

- Serial Schedules always gives correct result
- But there are some drawbacks
- In order to avoid that we introduced non serial schedules
- But not all non serial schedules are correct
- So we need a non serial schedule which always give a correct result or Equivalent to a
Serial Schedule = Serializable Schedule

3. Serializable Schedule

- Serializable Schedule is used to maintain the consistency of the database.
- It is mainly used in the Non-Serial scheduling to verify whether the scheduling will lead to any inconsistency or not.
- **The non-serial schedule is said to be in a serializable schedule only when it is equivalent to the serial schedules, for an n number of transactions.**
- Since concurrency is allowed in this case thus, multiple transactions can execute concurrently.
- These are of **two types**:

1. **Conflict Serializable** : A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

2. **View Serializable** : A Schedule is called view serializable if it is view equal to a serial schedule (no overlapping transactions).

When are two schedules considered equivalent?

- Result equivalence
- Conflict equivalence
- View equivalence

1. Result Equivalence

- If two schedules produce the same result after execution, they are said to be result equivalent.
- They may yield the same result for some value and different results for another set of values.
- That's why this equivalence is not generally considered significant.
- Eg :

S ₁
read_item(X); X:=X+10; write_item(X);

S ₂
read_item(X); X:=X*1.1; write_item(X);

For X=100: S₁ & S₂ are result equivalent
But for other values, not result equivalent
//S₁ and S₂ should not be considered equivalent

2. Conflict Equivalence

Conflict Operations

- Two schedules would be conflicting if they have the following properties –
 1. They belong to different transactions
 2. They access the same data item
 3. At least one of the operation is a “ write “ operation
- Two schedules having multiple transactions with conflicting operations are said to be **conflict equivalent** if and only if
 - **Both the schedules contain the same set of Transactions.**
 - **The order of conflicting pairs of operation is maintained in both the schedules.**

- Eg: Schedule S:

R1(X)	W2(X)	: Conflict
R2(X)	W1(X)	: Conflict
W1(X)	W2(X)	: Conflict
R1(X)	R2(X)	: No Conflict
W2(X)	W1(Y)	: No Conflict
R1(X)	W1(X)	: No Conflict

- Read Write Conflict

X=10;	
R1(X)	W2(X)//X=20
W2(X)//20	R1(X)
//T1 reads 10	//T2 reads 20

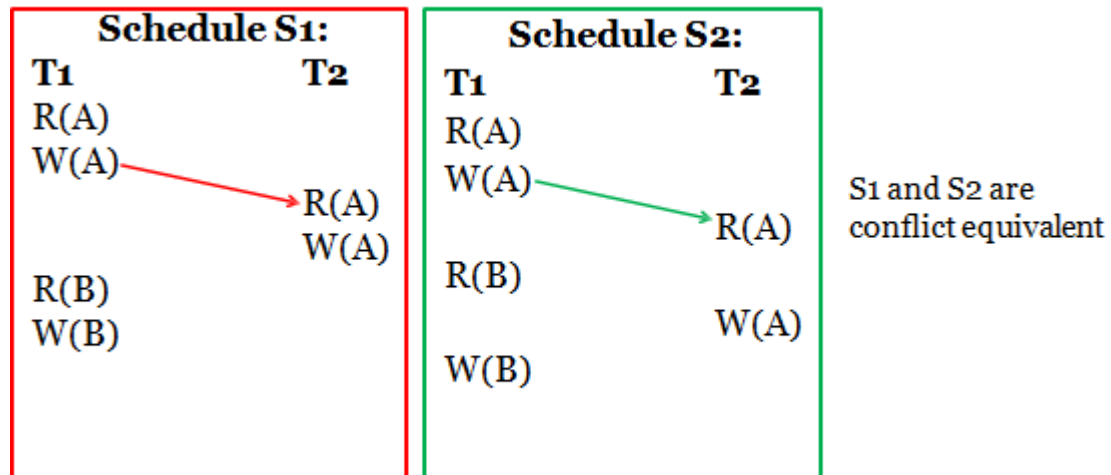
- Write Write Conflict

X=10;	
W1(X) //20	W2(X) //30
W2(X) //30	W1(X) //20
//Final Value 30	//Final Value 20

- Read Read Non Conflicting

X=10;	
R1(X)	R2(X)
R2(X)	R1(X)
//T1 & T2 reads 10	//T1 & T2 reads 10

Check whether two schedules are conflict equivalent or not



Two schedules are said to be conflict equivalent if the relative order of any two conflicting operations is the same in both schedules

Eg 2 :

Schedule S1:	
T1	T2
R(A)	
	R(B)
W(A)	
	W(B)

Schedule S2:	
T1	T2
	R(B)
R(A)	
	W(B)
W(A)	

No conflicting operations in S1 and S2 .
So they are conflict equivalent

Eg 3 :

Schedule S1:	
T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
R(B)	

Schedule S2:	
T1	T2
R(A)	
W(A)	
R(B)	
	R(B)
	W(B)

S1 and S2 are **not** conflict equivalent

PRECEDENCE GRAPH

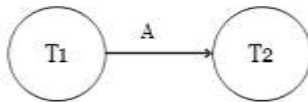
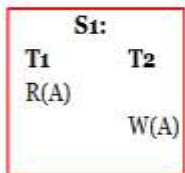
- To check a schedule is conflict serializable or not, we can use precedence graph
- **If the precedence graph is free from cycles(acyclic) then schedule is conflict serializable schedule**

Steps for precedence graph :

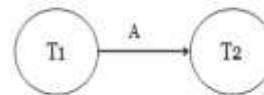
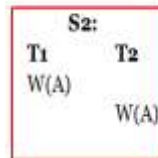
- For each transaction T, put a node or vertex in the graph.
- For each conflicting pair, put an edge from T_i to T_j .
- If there is a cycle in the graph then schedule is not conflict serializable else schedule is conflict serializable.

Rules for constructing Precedence Graph :

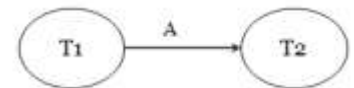
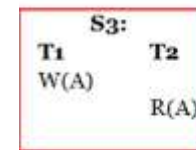
Rule 1



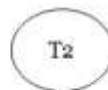
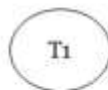
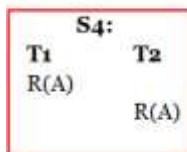
Rule 2



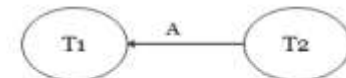
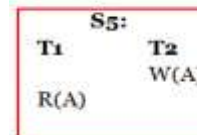
Rule 3



Rule 4



Rule 5



- It is a **directed Graph** $G = (V, E)$, which consists of a pair of a set of nodes and a set of directed edges

$$V = \{V1, V2, V3, ..., Vn\}$$

$$E = \{E1, E2, E3, ..., Em\}.$$

Where the set of nodes V are testing to retrieve identical data attribute through the transactions of a schedule and the set of edges E is regulated connectivity between a set of two nodes.

- **Nodes :** In the graph, for each transaction T_p the graph contains a single node.

So, In a schedule of a **precedence graph**, The total number of transactions will be similar to the total number of nodes.

- **Edges :** An edge is regulated connectivity between a set of two distinct transactions T_q and T_r and it shows in the format $T_q \rightarrow T_r$, where T_q is the beginning of the edge and T_r is the ending.

Algorithm for testing Conflict Serializability of a schedule

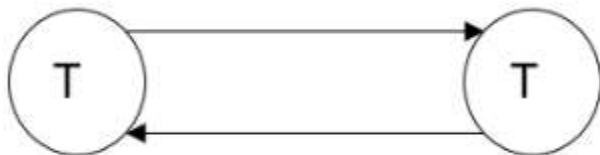
1. Create a node T , for each transaction participating in schedule S in the precedence graph.
2. For every condition in schedule S create an edge $T_p \rightarrow T_q$ in the precedence graph if a Transaction T_q implements a read_item (Z) after T_p implements a write_item (Z). It's a Read-Write conflict.
3. For every condition in schedule S create an edge $T_p \rightarrow T_q$ in the precedence graph if a Transaction T_q implements a write_item (Z) after T_i implements a read_item (Z). It's a Write-Read conflict.
4. For every condition in schedule S create an edge $T_p \rightarrow T_q$ in the precedence graph If a Transaction T_q implements a write_item (Z) after T_p implements a write_item (Z). It's a Write-Write conflict.
5. If and only if there is no cycle in the precedence graph, then the schedule S is Serializable.

Example:

Q1) Find the following Schedule S is conflict Serializable or not?

Transaction p	Transaction q
read (A)	
$A = A * 7$	
	read (A)
	$Temp = A + 45$
	$A = A + temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B = B - 32$	
write (B)	
	$B = B + temp$
	write (B)

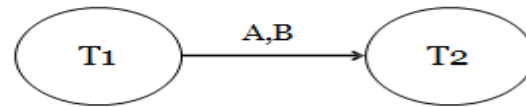
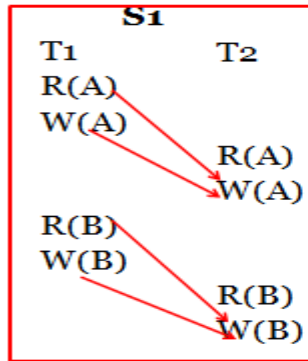
Solution :



In this precedence graph, by following accordingly to the Algorithm,

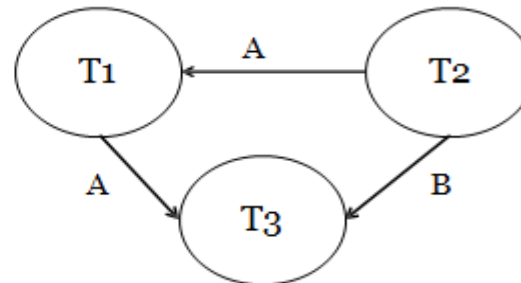
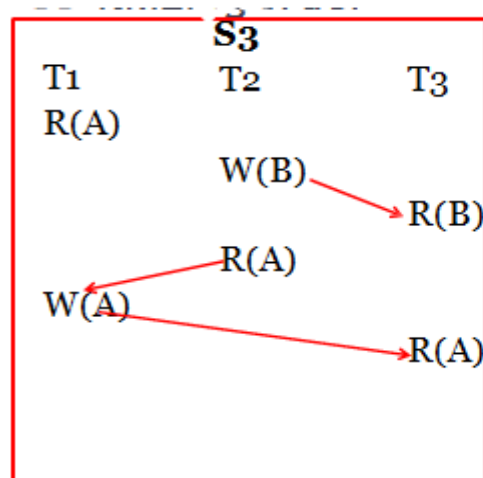
1. Transaction Tp implements reads A before Transaction Tq implements writes A, therefore the first arrow directed from Transaction Tp towards Transaction Tq
2. Transaction Tq reads B before Transaction Tp writes B, therefore the second arrow directed from Transaction Tq towards Transaction Tp.
3. Since from the above precedence graph it's clearly visible that the graph is cyclic, **therefore the schedule S is not conflicted Serializable.**

Eg 2 :



No Cycle in precedence graph so S1 is conflict serializable

Eg 3 :



NO Cycle in precedence graph
so S1 is conflict serializable.

3. View Equivalence

- Two schedules S1 and S2 are said to be view equivalent if for each data item,
 - If the initial read in S1 is done by T_i , then same transaction should be done in S2 also
 - If the final write in S1 is done by T_i , then in S2 also T_i should perform the final write
 - If T_i reads the item written by T_j in S1, then S2 also T_i should read the item written by T_j
- A non serial schedule is view serializable schedule if it is view equivalent to some of its serial schedule

Check if S1 is view serializable

S1	
T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

S2: T1T2	
T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

S1 and S2 is view
serializable
hence serializable

Initial read in S1 by T1 and S2 T2

Final write in S1 is by T2 AND S2 is by T2

Writer reader conflict in data item is same sequence in both S1 AND s2

Important Points

- **Every Conflict serializable schedule is View Serializable but vice versa is not true**
- **Every Conflict Serializable schedule is Serializable but vice versa is not true**
- **Every View Serializable schedule is serializable and vice versa is also true**

CONCURRENCY CONTROL PROTOCOLS

- Concurrency control protocols in DBMS ensure that multiple transactions can execute concurrently without interfering with each other and maintaining data integrity.
- These protocols aim to **prevent issues like lost updates, dirty reads, and non-repeatable reads.**
- They work by managing access to data resources and ensuring that transactions execute in a way that preserves the consistency of the database
- There are 3 types of Concurrency Control Protocols

Types of Concurrency Control Protocols

1. Lock-Based Protocols:

- These protocols use locks (shared and exclusive) to control access to data.
- A transaction needs to acquire a lock before accessing or modifying data.
- Shared locks allow multiple transactions to read the same data, while exclusive locks allow only one transaction to modify the data.

2. Timestamp-Based Protocols:

- Each transaction is assigned a timestamp when it is initiated.
- Transactions are compared based on their timestamps to resolve conflicts.
- Transactions with older timestamps generally have precedence when accessing data

LOCK BASED PROTOCOL

- A lock is a mechanism to control access to a data item
 - Ie, Locking protocol means, whenever a transaction wants to perform any operation(Read/Write) on any data item, first it should request for lock on that data item.
 - Once it acquire the lock then only it can perform the operation
 - **Locking mechanism is managed by Concurrency Control Subsystem**
-
- There are **Two type of Locks**
1. Shared Lock (S(A))
 2. Exclusive Lock (X(A))

1. SHARED LOCK

- It is also known as **Read only lock**
- In a shared lock, the data item can only read by the transaction
- It can be shared between the transactions, because when the transaction holds a lock, then it can't update the data on the data item
- Shared lock is requested using **lock –S** instruction

For eg :

Consider when two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account balance, shared lock prevent it until the reading process is over

**If a transaction wants to perform read operation on a data item,
it should apply shared lock on that item**

T1
S(A)
Read(A)

2. EXCLUSIVE LOCK

- In exclusive lock, data item can be both reads as well as written by the transaction
- This lock is exclusive , and in this lock , multiple transactions donot modify the same data simultaneously
- Transaction may unlock the data item after finishing the “ write “ operation so that the other transaction can acquire lock on that data item for its operations
- Exclusive lock is requested using **lock –X** instruction

For eg :

Consider when a transaction needs to update the account balance of a person . You can allow this transaction by placing X lock on it . Therefore when the second transaction wants to read or write exclusive lock prevent this operation

If a transaction acquire this lock,
it can perform read/write/both operation on that data item

T1
X(A)
Read(A)
Write(A)

Lock – Compatibility Matrix

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared lock on an item, but if any transaction holds an exclusive lock on the item no other transaction may hold any lock on the item
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released

	S	X
S	true	false
X	false	false

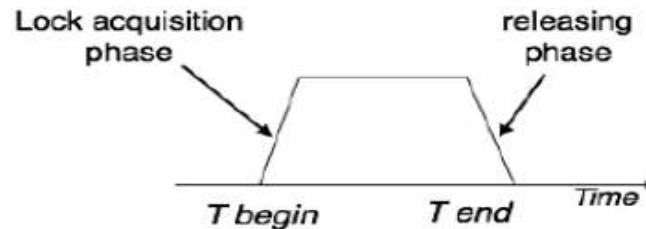
Problems with Simple Locking Protocol

- I. Sometimes does not guarantee Serializability
- II. May lead to deadlock

*To guarantee serializability, must follow some additional protocol concerning the positioning of **locking and unlocking** operation in every transaction*

TWO PHASE LOCKING (2PL)

- Two Phase Locking is also known as 2PL protocol / 2PL
- In this type of locking protocol, the transaction should acquire a lock after it releases one of its locks
- This locking protocol divides the execution phase of a transaction into three different parts
 - In the first phase, when the transaction begins to execute, it requires permission for the locks it needs
 - The second part is where the transaction obtains all the locks
 - when the transaction releases the first lock, the third phase starts
 - In the third phase , the transaction cannot demand any new locks. Instead it only releases the acquired locks



- If every individual transactions follows 2PL, then all schedules in which these transactions participate become Conflict Serializable
- 2PL Schedule are always Conflict Serializable Schedules

Two Phases of 2PL

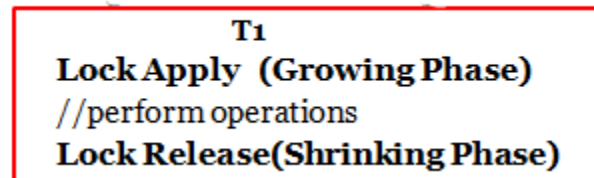
- A schedule is said to be in 2PL if all transactions perform locking and unlocking in 2 phases

1. Growing Phase

- New locks on data items can be acquired but none can be unlocked
- Only locking is allowed no unlocking

2. Shrinking Phase

- Existing locks may be released but no new locks can be acquired
- Only unlocking is allowed no locking



- Every transaction should first perform Growing phase and then Shrinking phase
- After Shrinking phase, Growing phase is not possible
- Growing phase is compulsory, but shrinking phase is optional
- 2PL schedules are always Conflict Serializable but vice versa is not true
- **Lock point:** The point at which a transaction acquire the last lock

Problem with Basic 2PL

1. Basic 2PL does not ensure Recoverability
2. Cascading rollbacks are possible
3. Deadlocks are possible

Variations of 2PL

1. Strict 2PL

- Follow basic 2PL + All exclusive locks should unlock only after commit operation
- Strict Schedule : A transaction is neither allowed to read/write a data item until the transaction that has written is committed

2. Rigorous 2PL

- More stronger than Strict 2PL
- Follow basic 2PL + All locks (both exclusive and shared locks) should unlock only after commit operation
- Every Rigorous 2PL is Strict 2PL but vice versa is not true

3. Conservative 2PL(C2PL)

- No Dealock
- Follow Basic 2PL + All transaction should obtain all lock they need before the transaction begins
- Release all lock after commit
- Recoverable, Serializable, Cascadeless, Deadlock Free
- Not practical to implement

SINo	Strict 2PL	Rigorous 2PL	Conservative 2PL
Benifits	<ul style="list-style-type: none"> • GeneratesConflict Serializable Schedules • ProduceStrict Schedules (hence it is recoverable and cascadeless) 	<ul style="list-style-type: none"> • GeneratesConflict Serializable Schedules • Produce Strict Schedules (hence it is recoverable and cascadeless) 	<ul style="list-style-type: none"> • Free from deadlock : <p>A transaction will begin its execution only if all locks are available so there is no chance of waiting for any resources during execution time (No Hold and Wait)</p>
Drawback	<ul style="list-style-type: none"> • Deadlock is possible 	<ul style="list-style-type: none"> • Deadlock is possible 	<ul style="list-style-type: none"> • Poor resource utilization • Concurrency is limited • Each transaction needs to declare all the data items that need to be read/write at beginning, which is not always possible

TIMESTAMP-BASED PROTOCOLS

- The timestamp based algorithm uses a timestamp to serialize the execution of concurrent transactions
- This protocol ensures that every conflicting read and write operations are executed in timestamp order
- The protocol uses the **System Time or Logical Count** as a timestamp
- The older transaction is always given priority in this method
- It uses system time to determine the time stamp of the transaction
- This is the most commonly used concurrency protocol

* *Lock based protocols help to manage the order between the conflicting transactions when they will execute*

* *Time stamp based protocols manage conflicts as soon as an operation is created*