DBMS | Concurrency Control - Introduction and ACID Properties

Concurrency Control deals with **interleaved execution** of more than one transaction. In the next article, we will see what is serializability and how to find whether a schedule is serializable or not.

What is Transaction?
A set of logically related operations is known as transaction. The main operations of a transaction are:

Read(A): Read operations Read(A) or R(A) reads the value of A from the database and stores it in a buffer in main memory.

Write (A): Write operation Write(A) or W(A) writes the value back to the database from buffer.

Let us take a debit transaction from an account which consists of following operations:

1. R(A);
2. A=A-1000;

3. W(A);

Assume A's value before starting of transaction is 5000.

- The first operation reads the value of A from database and stores it in a buffer.
- Second operation will decrease its value by 1000. So buffer will contain 4000.
- Third operation will write the value from buffer to database. So A's final value will be 4000.

But it may also be possible that transaction may fail after executing some of its operations. The failure can be because of **hardware, software or power** etc. For example, if debit transaction discussed above fails after executing operation 2, the value of A will remain 5000 in the database which is not acceptable by the bank. To avoid this, Database has two important operations:

Commit: After all instructions of a transaction are successfully executed, the changes made by transaction are made permanent in the database.

Rollback: If a transaction is not able to execute all operations successfully, all the changes made by transaction are undone.

Properties of a transaction

- **Atomicity**: By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. 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'.
  Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X − 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of **T1** but before completion of **T2**.(say, after **write(X)** but before **write(Y)**), then amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the

transaction must be executed in entirety in order to ensure correctness of database state.

- **Consistency**: This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above,
  The total amount before and after the transaction must be maintained.
  Total **before T** occurs = **500 + 200 = 700**.
  Total **after T occurs** = **400 + 300 = 700**.
  Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result, T is incomplete.
- **Isolation**: This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the 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.
  Let **X**= 500, **Y** = 500.
  Consider two transactions **T** and **T''.**

T	T''
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y – 50 Write	Read (X) Read (Y) Z: = X + Y Write (Z)

  Suppose **T** has been executed till **Read (Y)** and then **T''** starts. As a result , interleaving of operations takes place due to which **T''** reads correct value of **X** but incorrect value of **Y** and sum computed by
  T'': (X+Y = 50, 000+500=50, 500)
  is thus not consistent with the sum at end of transaction: **T: (X+Y = 50, 000 + 450 = 50, 450)**.
  This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.
- **Durability:** 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.

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, acts in isolation from other operations and updates that it makes are durably stored.

DBMS | Concurrency Control | Schedule and Types of Schedules

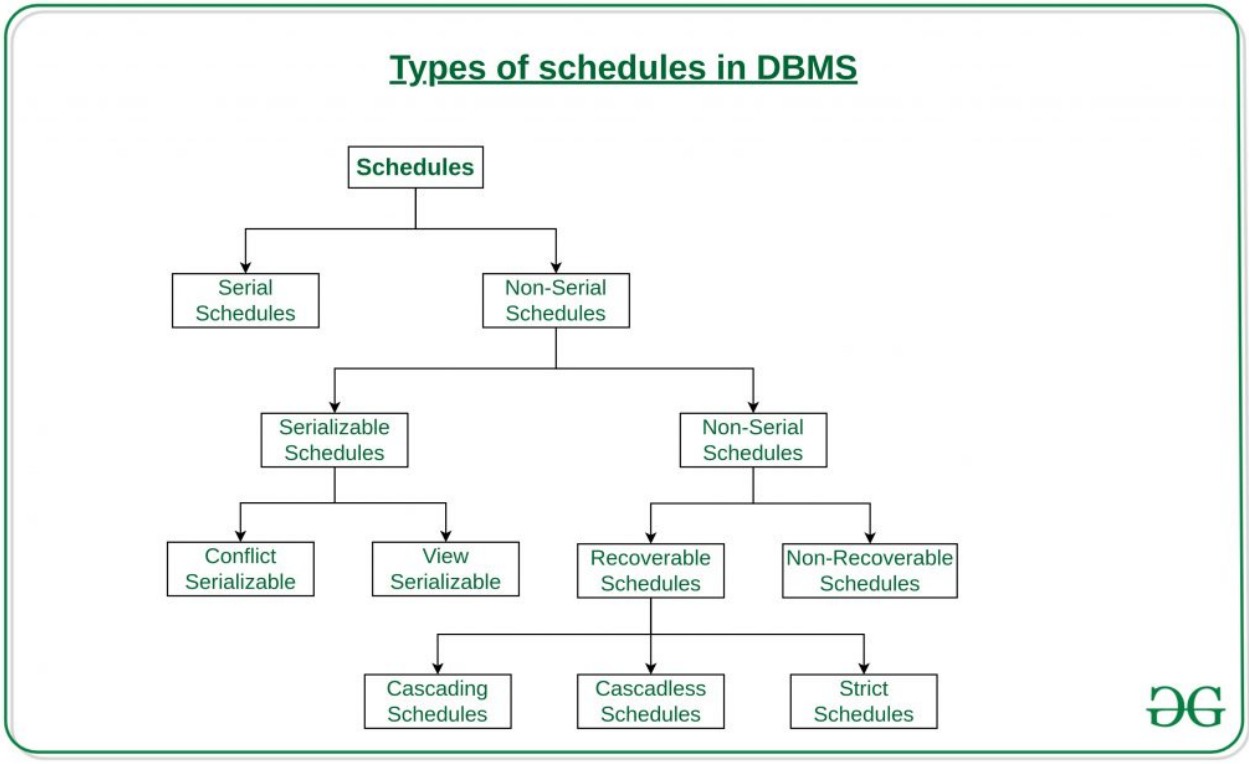
What is a Schedule?

A schedule is a series of operations from one or more transactions. Schedules are used to resolve conflicts between the transactions.

Schedule, as the name suggests, is a process of lining the transactions and executing them one by one. 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, Scheduling is brought into play and the transactions are timed accordingly.

Types of Schedules?

Lets discuss various types of schedules.



1. **Serial Schedules:**
   Schedules in which the transactions are executed non-interleaved, i.e., a serial schedule is one in which no transa arts until a running transaction has

ended are called serial schedules.

Example: Consider the following schedule involving two transactions T_1 and T_2.

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 T_1 —> T_2

2. **Non-Serial Schedule:** This is a type of Scheduling where the operations of multiple transactions are interleaved. This might lead to a rise 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. It can be of two types namely, Serializable and Non-Serializable Schedule.

   The Non-Serial Schedule can be divided further into Serializable and Non-Serializable.

   a. **Serializable:** This 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. On the other hand, a serial schedule does not need the serializability because it follows a transaction only when the previous transaction is complete. 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. A serializable schedule helps in improving both resource utilization and CPU throughput. These are of two types:

      1. **Conflict Serializable:** A schedule is called conflict serializable if it can be transformed into a seria[illegible]ule by swapping non-conflicting operations. Two operati[illegible] said to be conflicting if all conditions

satisfy:

- They belong to different transactions
- They operate on the same data item
- At Least one of them is a write operation

2. **View Serializable:** A Schedule is called view serializable if it is view equal to a serial schedule (no overlapping transactions). A conflict schedule is a view serializable but if the serializability contains blind writes, then the view serializable does not conflict serializable.

b. **Non-Serializable:** The non-serializable schedule is divided into two types, Recoverable and Non-recoverable Schedule.

1. **Recoverable Schedule:** Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, 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.
Example 1:

```
S1: R1(x), W1(x), R2(x), R1(y), R2(y),

          W2(x), W1(y), C1, C2;
```

Given schedule follows order of **Ti->Tj => C1->C2**. Transaction T1 is executed before T2 hence there is no chances of conflict occur. R1(x) appears before W1(x) and transaction T1 is committed before T2 i.e. completion of first transaction performed first update on data item x, hence given schedule is recoverable.

Example 2: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
commit	
	commit

This is a recoverable schedule since T_1 commits before T_2, that makes the value read by T_2 correct.

There can be three types of recoverable schedule:

a. **Cascading Schedule:** When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or cascading abort. Example:

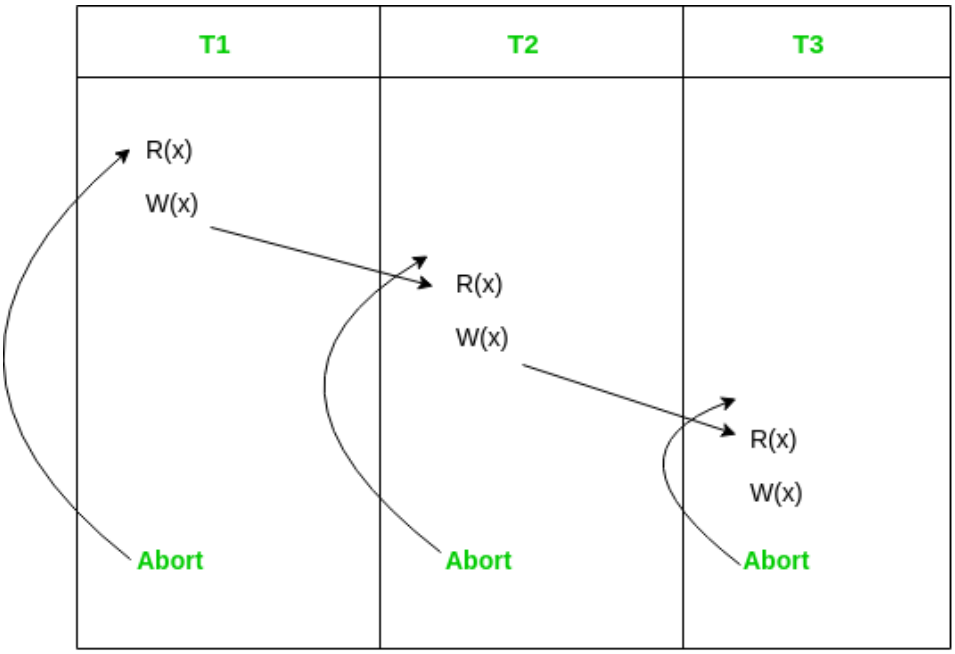


Figure - Cascading Abort

b. **Cascadeless Schedule:** Also called Avoids cascading aborts/rollbacks (ACA). Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

In other words, if some transaction T_j wants to read value updated or written by some other transaction T_i, then the commit of T_j must read it after the commit of T_i.

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
commit	

T_1	T_2
	R(A)
	commit

This schedule is cascadeless. Since the updated value of A is read by T_2 only after the updating transaction i.e. T_1 commits.

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
abort	
	abort

It is a recoverable schedule but it does not avoid cascading aborts. It can be seen that if T_1 aborts, T_2 will have to be aborted too in order to maintain the correctness of the schedule as T_2 has already read the uncommitted value written by T_1.

c. **Strict Schedule:** A schedule is strict if for any two transactions T_i, T_j, if a write operation of T_i precedes a conflicting operation of T_j (either read or write), then the commit or abort event of T_i also precedes that conflicting operation of T_j.
In other words, T_j can read or write updated or written value of T_i only after T_i commits/aborts.

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
	R(A)
W(A)	
commit	
	W(A)
	R(A)
	commit

This is a strict schedule since T_2 reads and writes A which is written by T_1 only after the commit of T_1.

2. **Non-Recoverable Schedule:** The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. T2 commits. But later on, T1 fails. So we have to rollback T1. Since T2 has read the value written by T1, it should also be rollbacked. But we have already committed that. So this schedule is irrecoverable schedule. When Tj is reading the value updated by Ti and Tj is committed before committing of Ti, the schedule will be irrecoverable.

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
		Commit;		
Failure Point				
Commit;				

Note - It can be seen that:

1. Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
2. Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.
3. Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.

- DBMS | Conflicting Serializability

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

Conflicting operations: Two operations are said to be conflicting if all conditions satisfy:

- They belong to different transactions
- They operate on the same data item
- At Least one of them is a write operation

Example: -

- **Conflicting** operations pair ($R_1(A)$, $W_2(A)$) because they belong to two different transactions on same data item A and one of them is write operation.
- Similarly, ($W_1(A)$, $W_2(A)$) and ($W_1(A)$, $R_2(A)$) pairs are also **conflicting**.
- On the other hand, ($R_1(A)$, $W_2(B)$) pair is **non-conflicting** because they operate on different data item.
- Similarly, (($W_1(A)$, $W_2(B)$) pair is **non-conflicting.**

Consider the following schedule:

S1: $R_1(A)$, $W_1(A)$, $R_2(A)$, $W_2(A)$, $R_1(B)$, $W_1(B)$, $R_2(B)$, $W_2(B)$

If O_i and O_j are two operations in a transaction and $O_i < O_j$ (O_i is executed before O_j), same order will follow in schedule as well. Using this property, we can get two transactions of schedule S1 as:

T1: $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$

T2: $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$

Possible Serial Schedules are: T1->T2 or T2->T1

-> **Swapping non-conflicting operation**s $R_2(A)$ and $R_1(B)$ in S1, the schedule becomes,

S11: $R_1(A)$, $W_1(A)$, $R_1(B)$, **$W_2(A)$,** $R_2(A)$, **$W_1(B)$,** $R_2(B)$, $W_2(B)$

-> Similarly, s**wapping non-conflicting operations** $W_2(A)$ and $W_1(B)$ in S11, the schedule becomes,

S12: $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$, $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$

S12 is a serial schedule in which all operations of T1 are performed before starting any operation of T2. Since S has been transformed into a serial schedule S12 by swapping non-conflicting operations of S1, S1 is conflict serializable.

Let us take another Schedule:

S2: $R_2(A)$, $W_2(A)$, $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$, $R_2(B)$, $W_2(B)$

Two transactions will be:

T1: $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$

T2: $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$

Possible Serial Schedules are: T1->T2 or T2->T1

Original Schedule is:

S2: $R_2(A)$, $W_2(A)$, **$R_1(A)$,** $W_1(A)$, $R_1(B)$, $W_1(B)$, **$R_2(B)$,** $W_2(B)$

Swapping non-conflicting operations $R_1(A)$ and $R_2(B)$ in S2, the schedule becomes,

S21: $R_2(A)$, $W_2(A)$, $R_2(B)$, **$W_1(A)$,** $R_1(B)$, $W_1(B)$, $R_1(A)$, **$W_2(B)$**

Similarly, swapping non-conflicting operations $W_1(A)$ and $W_2(B)$ in S21, the schedule becomes,

S22: $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$, $R_1(B)$, $W_1(B)$, $R_1(A)$, $W_1(A)$

In schedule S22, all operations of T2 are performed first, but operations of T1 are not in order (order should be $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$). So S2 is not conflict serializable.

Conflict Equivalent: Two schedules are said to be conflict equivalent when one can be transformed to another by swapping non-conflicting operations. In the example discussed above, S11 is conflict equivalent to S1 (S1 can be converted to S11 by swapping non-conflicting operations). Similarly, S11 is conflict equivalent to S12 and so on.

Note 1: *Although S2 is not conflict serializable, but still it is conflict equivalent to S21 and S21 because S2 can be converted to S21 and S22 by swapping non-conflicting operations.*

Note 2: *The schedule which is conflict serializable is always conflict equivalent to one of the serial schedule. S1 schedule discussed above (which is conflict serializable) is equivalent to serial schedule (T1->T2).*

Question: Consider the following schedules involving two transactions. Which one of the following statement is true?

S1: $R_1(X)$ $R_1(Y)$ $R_2(X)$ $R_2(Y)$ $W_2(Y)$ $W_1(X)$

S2: $R_1(X)$ $R_2(X)$ $R_2(Y)$ $W_2(Y)$ $R_1(Y)$ $W_1(X)$

- Both S1 and S2 are conflict serializable
- Only S1 is conflict serializable
- Only S2 is conflict serializable
- None

[GATE 2007]

Solution: Two transactions of given sche re:

```
T1: R1(X) R1(Y) W1(X)

T2: R2(X) R2(Y) W2(Y)
```

Let us first check serializability of S1:

```
S1: R1(X) R1(Y) R2(X) R2(Y) W2(Y) W1(X)
```

To convert it to a serial schedule, we have to swap non-conflicting operations so that S1 becomes equivalent to serial schedule T1->T2 or T2->T1. In this case, to convert it to a serial schedule, we must have to swap $R_2(X)$ and $W_1(X)$ but they are conflicting. So S1 can't be converted to a serial schedule.

Now, let us check serializability of S2:

```
S2: R1(X) R2(X) R2(Y) W2(Y) R1(Y) W1(X)
```

Swapping non conflicting operations $R_1(X)$ and $R_2(X)$ of S2, we get

```
S2': R2(X) R1(X) R2(Y) W2(Y) R1(Y) W1(X)
```

Again, swapping non conflicting operations $R_1(X)$ and $R_2(Y)$ of S2', we get

```
S2'': R2(X) R2(Y) R1(X) W2(Y) R1(Y) W1(X)
```

Again, swapping non conflicting operations $R_1(X)$ and $W_2(Y)$ of S2'', we get

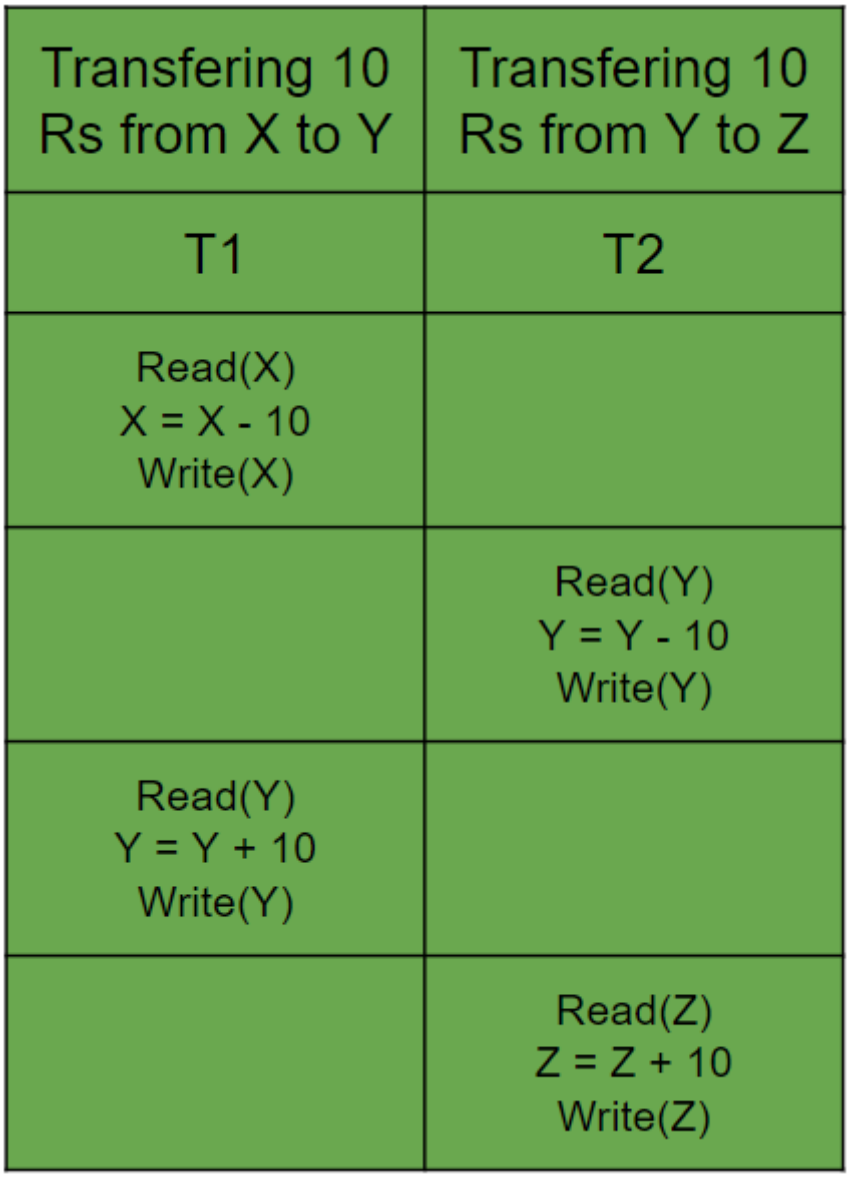
```
S2''': R2(X) R2(Y) W2(Y) R1(X) R1(Y) W1(X)
```

which is equivalent to a serial schedule T2->T1.

So, **correct option is C**. Only S2 is conflict serializable.

- View Serializability

The concurrent execution is allowed into DBMS so that the execution process is not slowed down, and the resources can be shifted when and where necessary. But the concurrency also creates another problem of resource allocation. If any ongoing transaction is under progress and the interleaving occurs in-between, then this may lead to inconsistency of the data. To maintain consistency, the interleaving must be tracked and checked. Isolation is one factor that must be kept in mind. It signifies whether all the individual transactions are run in an isolation manner or not, even when concurrency and interleaving of the process are allowed. Let's look at the below transactions T1 and T2.



Transfering 10 Rs from X to Y	Transfering 10 Rs from Y to Z
T1	T2
Read(X) X = X - 10 Write(X)	
	Read(Y) Y = Y - 10 Write(Y)
Read(Y) Y = Y + 10 Write(Y)	
	Read(Z) Z = Z + 10 Write(Z)

The impact of the interleaving schedule must be such that T1 is executed before T2(T1T2) or T2 is executed before T1(T2T1). The execution must be equivalent to this condition. This is known as serializability. This is used to verify whether an interleaving schedule is valid or not. These are of two types:

1. View Serializability

2. Conflict Serializability

Let's talk about View Serilizabilty:
Considering the above transactions, to check for View Serializability we need to arrange for both the sequences i.e., T1T2 and T2T1 and check whether the original sequence is equivalent to the serializable sequence or not.
Let's generate T1T2 and check whether it is view equivalent of the table:

```
Read(X)
X = X - 10
Write(X)
Read(Y)
```

```
Y = Y + 10
Write(Y)

Read(Y)
Y = Y - 10
Write(Y)
Read(Z)
Z = Z + 10
Write(Z)
```


To check for view serializability, we need to check these three conditions for every data item as in X, Y and Z:

1. Initial Read

2. Updated Read

3. Final Write

Initial Read: This specifies that the initial read on every data item must be done by the same transaction, in a serialized manner.
Let's check for T1T2:
For X, this is valid as in both the cases X is read by T1 first.
For Y, this condition fails, as in the serial case T1 reads Y first, but in the original table, it is first read by Y.
So, in this case, the view equivalence fails. Hence no more conditions are needed to be checked for T1T2.

Let's check for T2T1:

```
Read(Y)
Y = Y - 10
Write(Y)
Read(Z)
Z = Z + 10
Write(Z)

Read(X)
X = X - 10
Write(X)
Read(Y)
Y = Y + 10
Write(Y)
```

For Y, this is valid as in both the cases Y is read by T2 first. We don't need to check for X and Z as they are not shared by both the transactions.

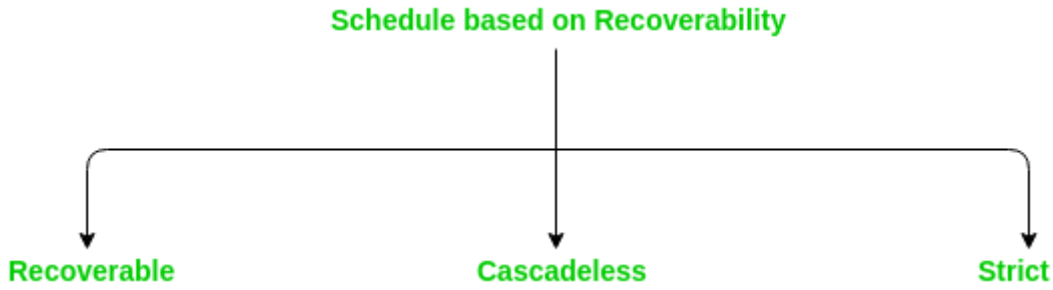
Updated Read: This says that when a transaction is reading a value that is written by another transaction, then the sequence must be the same. We don't need to check for X and Z as they are not sharing. In the case of Y T2T1, Y is written first then T1 reads it. Hence the condition is valid.

Final Write: Let's check for the final write for both the interleaved schedule and the serial schedule. We see that in both the case of T2T1, Y is written in a serial manner.

Therefore, we can conclude that it is viewed serializable in accordance with T2T1.

- DBMS | Types of Recoverable Schedules

Generally, there are three types of Recoverable schedule given as follows:



Let's discuss each one of these in detail.

1. **Recoverable Schedule:** A schedule is said to be recoverable if it is recoverable as the name suggest. Only reads are allowed before write operation on the same data. Only reads (Ti->Tj) is permissible.

   Example:

   ```
   S1: R1(x), W1(x), R2(x), R1(y), R2(y),
               W2(x), W1(y), C1, C2;
   ```

   Given schedule follows order of **Ti->Tj => C1->C2**. Transaction T1 is executed before T2 hence there is no chances of conflict occur. R1(x) appears before W1(x) and transaction T1 is committed before T2 i.e. completion of first transaction performed first update on data item x, hence given schedule is recoverable.

   Lets see example of **unrecoverable schedule** to clear the concept more:

```
S2: R1(x), R2(x), R1(z), R3(x), R3(y), W1(x),

      W3(y), R2(y), W2(z), W2(y), C1, C2, C3;
```

Ti->Tj => C2->C3 but W3(y) executed before W2(y) which leads to conflicts thus it must be committed before T2 transaction. So given schedule is unrecoverable. if **Ti->Tj => C3->C2** is given in schedule then it will become recoverable schedule.

Note: A committed transaction should never be rollback. It means that reading a value from the uncommitted transaction and commits it will enter the current transaction into the inconsistent or unrecoverable state this is called *Dirty Read problem*.

Example:

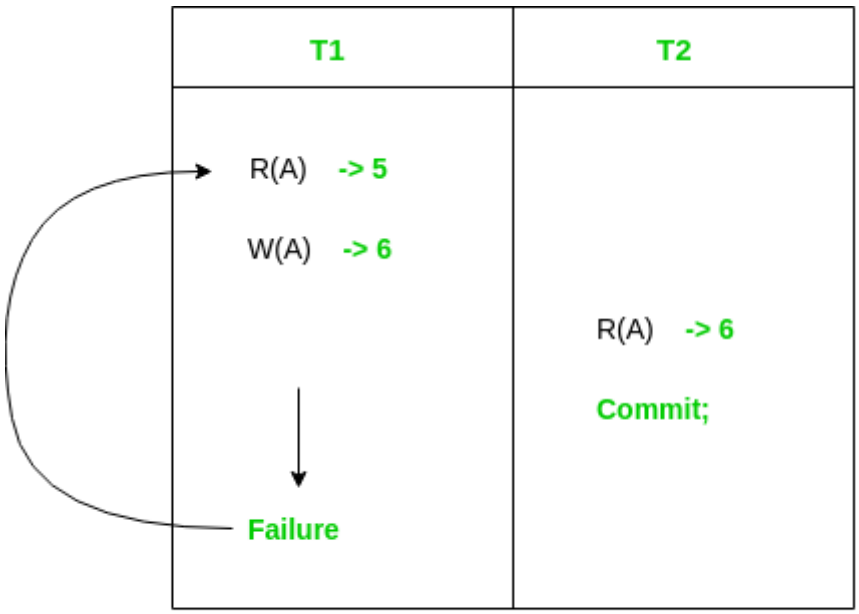


Figure - Dirty Read Problem

2. **Cascadeless Schedule:** When no **read** or **write-write** occurs before execution of transaction then corresponding schedule is called cascadeless schedule.

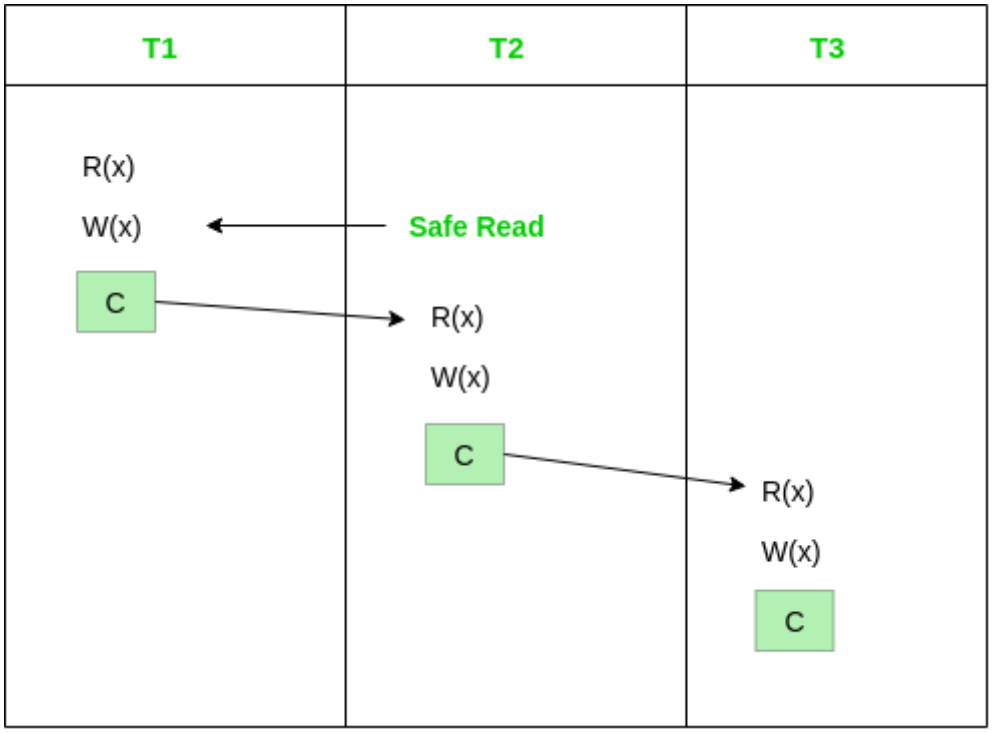
Example:

```
S3: R1(x), R2(z), R3(x), R1(z), R2(y), R3(y), W1(x), C1,

      W2(z), W3(y), W2(y), C3, C2;
```

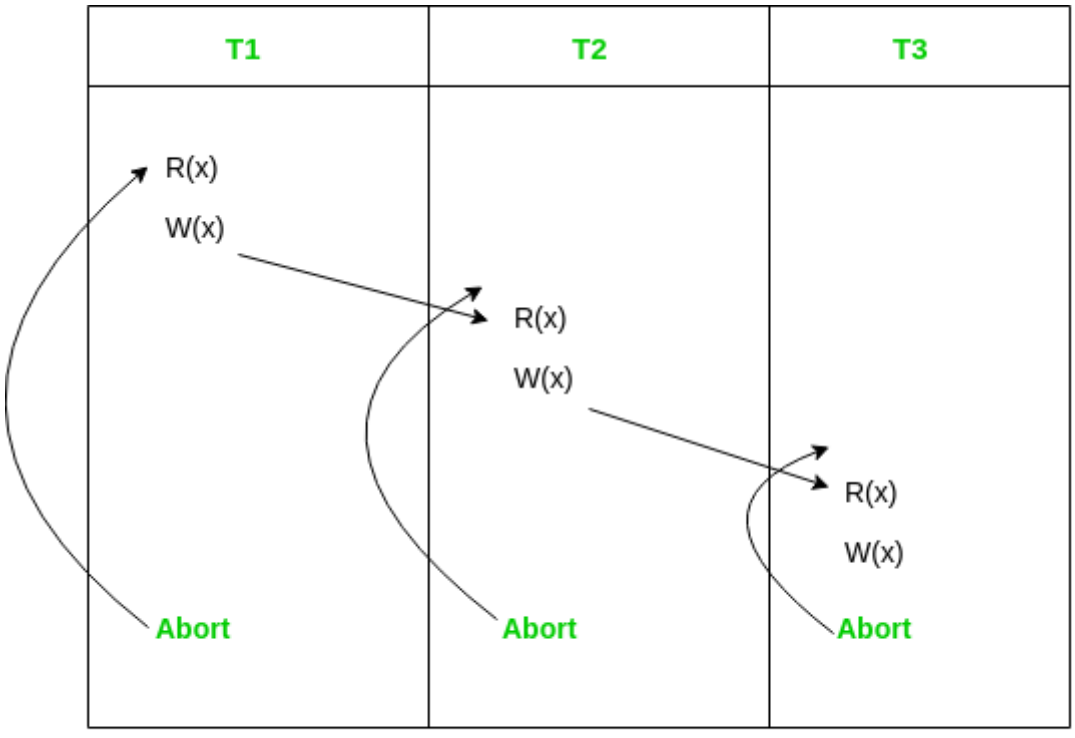
In this schedule **W3(y)** and **W2(y)** overwrite conflicts and there is no read, therefore given schedule is cascadeless schedule.

Special Case: A committed transaction desired to abort. As given below all the transactions are reading committed data hence it's cascade less schedule.



Cascading Abort: Cascading Abort can also be rollback. If transaction T1 abort as T2 read data that written by T1 which is not committed. Hence it's cascading rollback.

Example:



Fi cading Abort

3. **Strict Schedule:** If schedule contains no **read** or **write** before commit then it is known as strict schedule. Strict schedule is strict in nature.

   Example:

   ```
   S4: R1(x), R2(x), R1(z), R3(x), R3(y),

           W1(x), C1, W3(y), C3, R2(y), W2(z), W2(y), C2;
   ```

   In this schedule no read-write or write-write conflict arises before commit hence its strict schedule:

T1	T2	T3
R1(x)		
	R2(x)	
R1(z)		
		R3(x)
		R3(y)
W1(x)		
C1;		
		W3(y)
		C3;
	R2(y)	
	W2(z)	
	W2(y)	
	C2;	

Figure - Strict Schedule

Corelation between Strict, Cascadeless and Recoverable schedule:

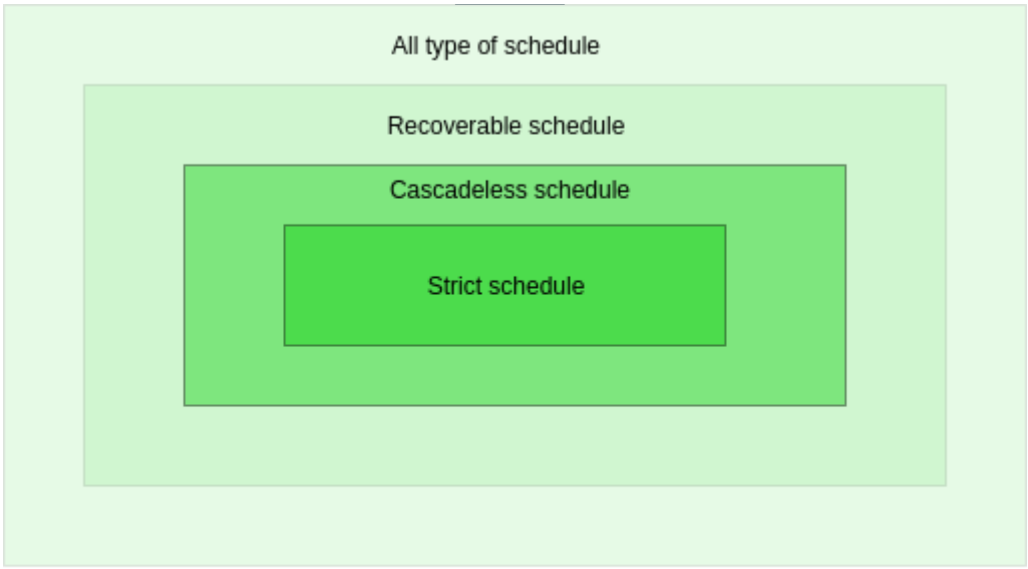


Figure - Venn diagram of these schedules

From above figure:

1. Strict schedules are all recoverable and cascadeless schedules
2. All cascadeless schedules are recoverable

Rollback and its types in DBMS

What is a Rollback?

A Rollback refers to an operation in which the database is brought back to some previous state, inorder to recover the loss of data or to overcome an unexpected situation. A rollback not only recovers the database to a stable position but also helps to maintain the integrity of the database.

A transaction may not execute completely due to hardware failure, system crash or software issues. In that case, we have to roll back the failed transaction. But some other transaction may also have used values produced by the failed transaction. So we have to roll back those transactions as well.

Types of Rollback?

1. **Cascading Recoverable Rollback:** The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. But later on, T1 fails. So we have to rol 1. Since T2 has read the value written by

T1, it should also be rollbacked. As it has not committed, we can rollback T2 as well. So it is recoverable with cascading rollback. Therefore, if Tj is reading value updated by Ti and commit of Tj is delayed till commit of Ti, the schedule is called recoverable with cascading rollback.

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
Failure Point				
Commit;				
		Commit;		

2. **Cascadeless Recoverable Rollback:** The table below shows a schedule with two transactions, T1 reads and writes A and commits and that value is read by T2. But if T1 fails before commit, no other transaction has read its value, so there is no need to rollback other transaction. So this is a Cascadeless recoverable schedule. So, if Tj reads value updated by Ti only after Ti is committed, the schedule will be cascadeless recoverable.

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
Commit;				
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
		Commit;		

- Dirty Read Problem and How to avoid it in DBMS

What is a Dirty Read Problem?

When a Transaction reads data from un ed write in another transaction, then it is

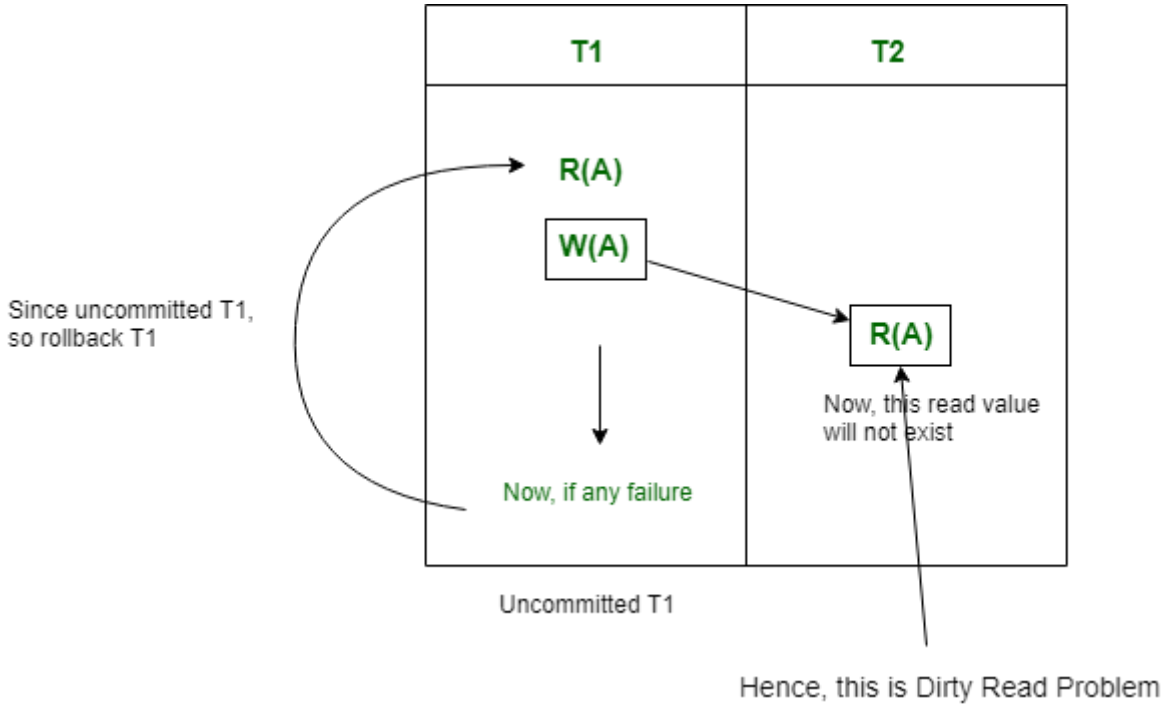
known as *Dirty Read*. If that writing transaction failed, and that written data may be updated again. Therefore, this causes *Dirty Read Problem*.

In other words,

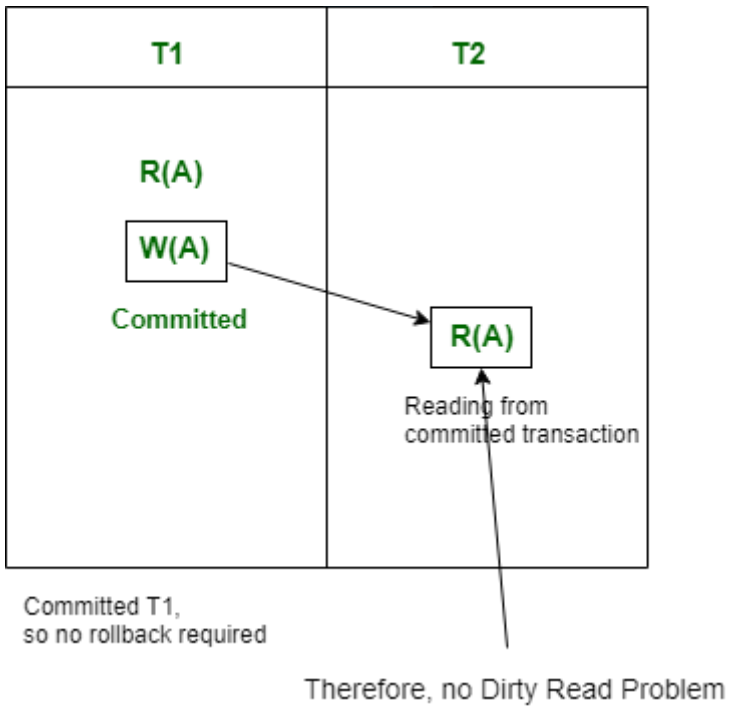
> Reading the data written by an uncommitted transaction is called as dirty read.

It is called dirty read because there is always a chance that the uncommitted transaction might roll back later. Thus, the uncommitted transaction might make other transactions read a value that does not even exist. This leads to inconsistency of the database.

For example, let's say transaction 1 updates a row and leaves it uncommitted, meanwhile, Transaction 2 reads the updated row. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.



Note that there is no Dirty Read problem, is a transaction is reading from another committed transaction. So, no rollback required.

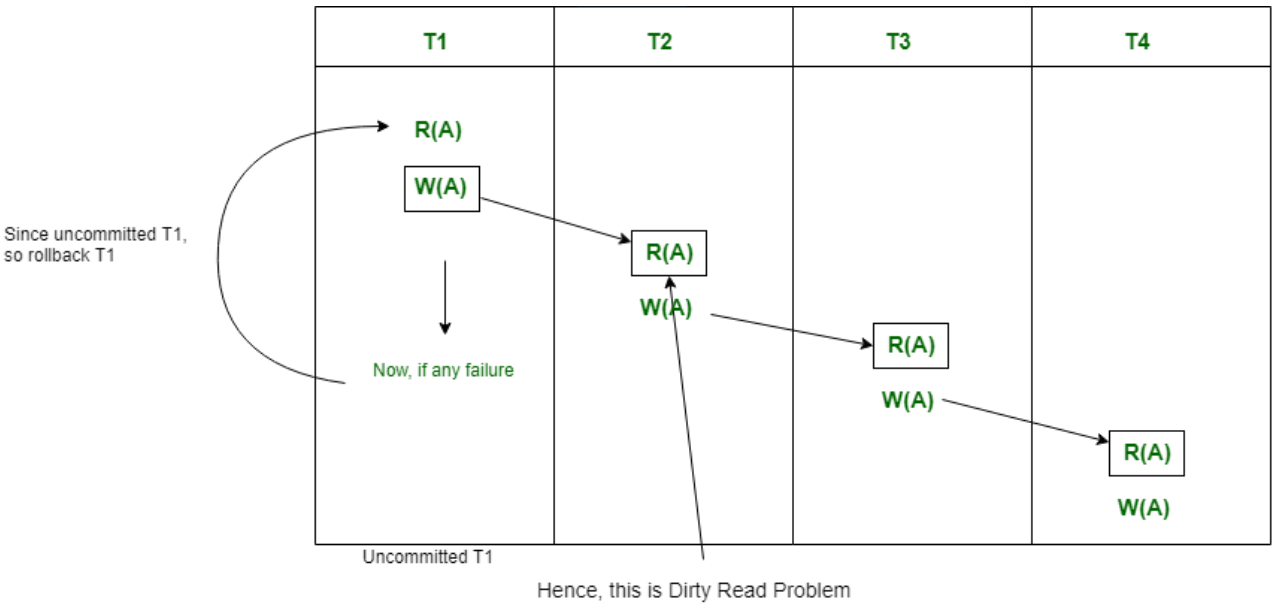


What does Dirty Read Problem result into?

Dirty Read Problem, in DBMS, results in the occurrence of Cascading Rollbacks.

Cascading Rollback: If in a schedule, failure of one transaction causes several other dependent transactions to rollback or abort, then such a schedule is called as a Cascading Rollback or Cascading Abort or Cascading Schedule. It simply leads to the wastage of CPU time.
These Cascading Rollbacks occur because of **Dirty Read problems**.

For example, transaction T1 writes uncommitted x that is read by Transaction T2.
Transaction T2 writes uncommitted x that is read by Transaction T3.
Suppose at this point T1 fails.
T1 must be rolled back since T2 is dependent on T1, T2 must be rolled back, and since T3 is dependent on T2, T3 must be rolled back.



Because of T1 rollback, all T2, T3, and T4 should also be rollback (Cascading dirty read problem).

This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks is called **Cascading rollback**.

How to avoid Dirty Read Problem?

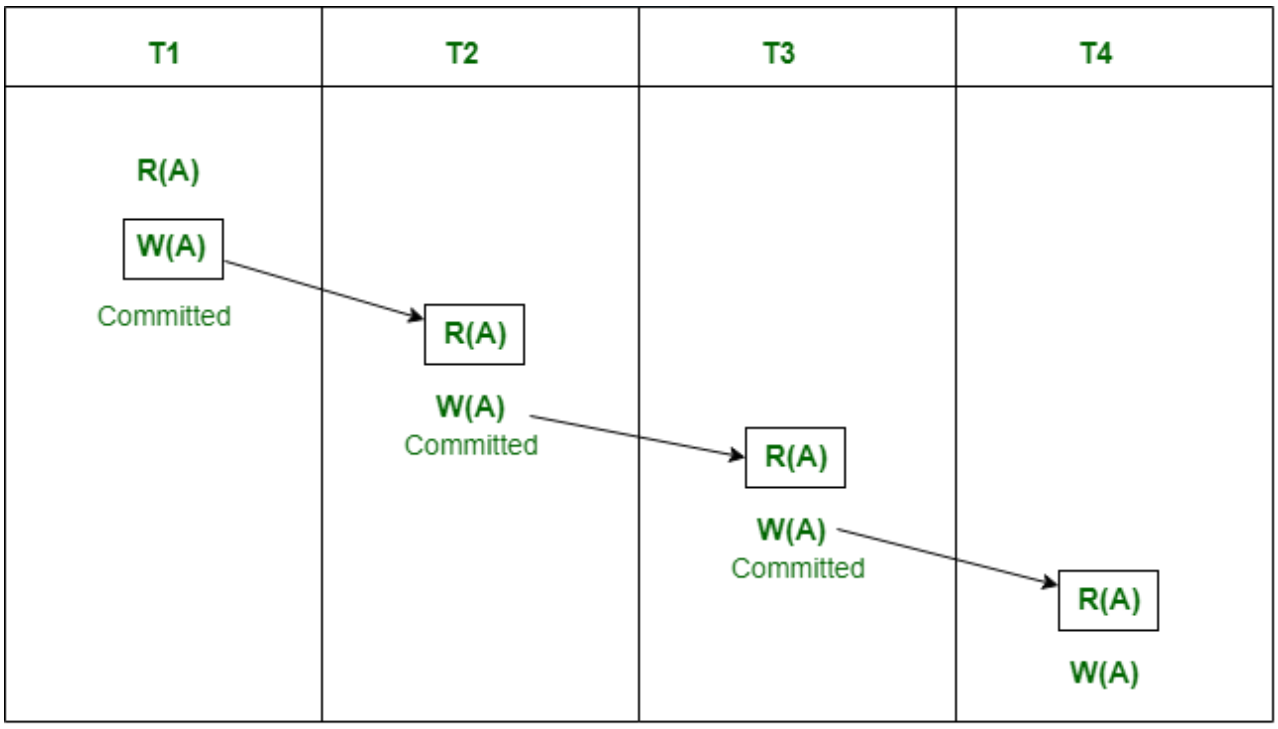
Dirty Read Problem can be avoided with the help of Cascadeless Schedule.

Cascadeless Schedule: This schedule avoids all possible *Dirty Read Problem*.

In Cascadeless Schedule, if a transaction is going to perform read operation on a value, it has to wait until the transaction who is performing write on that value commits. That means there must not be **Dirty Read**. Because Dirty Read Problem can cause *Cascading Rollback*, which is inefficient.

Cascadeless Schedule avoids cascading aborts/rollbacks (ACA). Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

In other words, if some transaction Tj wants to read value updated or written by some other transaction Ti, then the commit of Tj must read it after the commit of Ti.



No Dirty Read problem

Note that Cascadeless schedule allows only committed read operations. However, it allows uncommitted to write operations.

Also note that Cascadeless Schedules are always recoverable, but all recoverable transactions may not be Cascadeless Schedule.

- Two Phase Locking Protocol

Now, recalling where we last left off, there are two types of Locks available **Shared S(a)** and **Exclusive X(a)**. Implementing this lock system without any restrictions gives us the Simple Lock based protocol (or *Binary Locking*), but it has its own disadvantages, they **does not guarantee Serializability**. Schedules may follow the preceding rules but a non-serializable schedule may result.

To guarantee serializability, we must follow some additional protocol *concerning the positioning of locking and unlocking operations* in every transaction. This is where the concept of Two-Phase Locking(2-PL) comes in the picture, 2-PL ensures serializability. Now, let's dig deep!

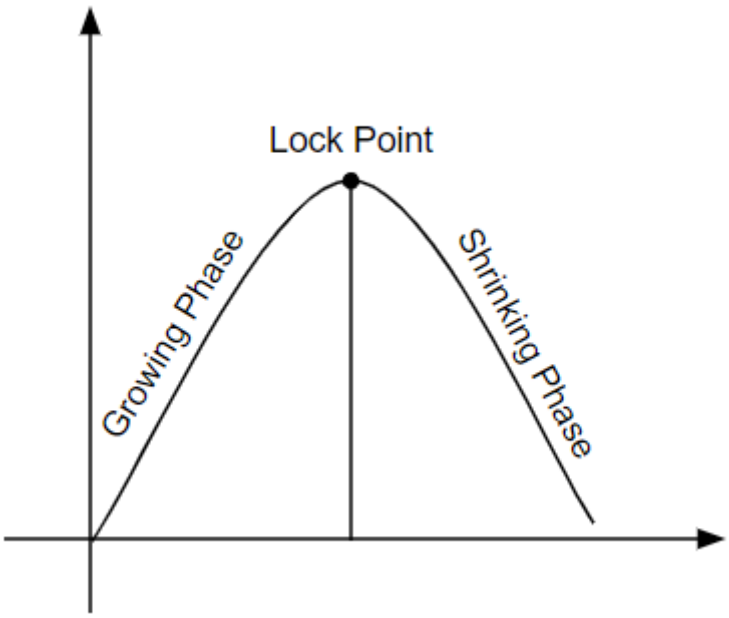
Let's get familiar with the two common locks which are used and some terminology followed in this protocol.

- **Shared Lock (S):** It is also known as Read-only lock. As the name suggests it can be shared between transactions because while holding this lock the transaction does not have the permission to update data on the data item. S-lock is requested using lock-S instruction.
- **Exclusive Lock (X):** In this lock, the data items can be both read and written. This is Exclusive and cannot be held simultaneously on the same data item. X-lock is requested using lock-X instruction.

Two Phase Locking -

A transaction is said to follow Two Phase Locking protocol if Locking and Unlocking can be done in two phases.

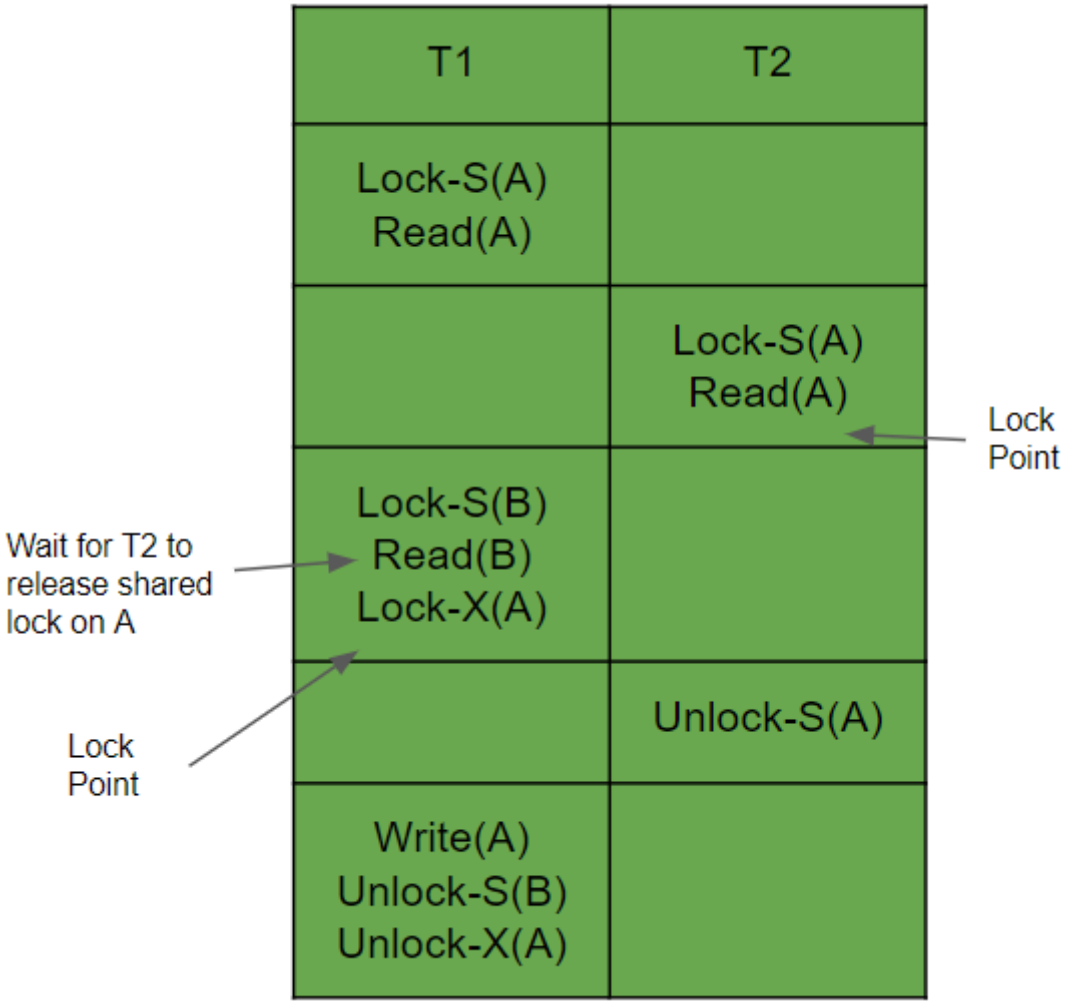
1. **Growing Phase:** New locks on data items may be acquired but none can be released.
2. **Shrinking Phase:** Existing locks may be released but no new locks can be acquired.



LOCK POINT: The Point at which the growing phase ends, i.e., when the transaction takes the final lock it needs to carry on its work. Now look at the schedule, you'll surely understand.

Note - If lock conversion is allowed, then upgrading of lock(from S(a) to X(a)) is allowed in Growing Phase and downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

Let's see a transaction implementing 2-PL.



This is just a skeleton transaction which shows how unlocking and locking work with 2-PL. Note for:
For Transaction T1, the growing phase is marked before the Lock point of T1, followed by its shrinking phase.
For Transaction T2, the growing phase is marked before the Lock point of T2, followed by its shrinking phase.

This transaction also takes care of Conflict Serializability as the transactions do not conflict with each other. The use of lock mechanism and lock point avoids the conflict. Since the transaction is conflict serializable, it automatically ensures view serializability. In the above example, T2T1 is conflict equivalent to this schedule. To check this let's draw the precedence graph.

Since there is no cycle involved, it ensures the serializability.

Problems with Basic 2PL: Deadlock in 2-PL - Consider this simple example, it will be easy to understand.Say we have two transactions T_1 and T_2.

```
Schedule:   T1: Lock-X1(A)   T2: Lock-X2(B)   T1: Lock-X1(B)   T2: Lock-
```

In this T1: Lock-X_1(B) has to wait for T2 and T2: Lock-X_2(A) has to wait for T1.

between Write(A) of T1 and Read(A) of T2, leading to irrecoverability.

T1	T2
Lock-X(A)	
Read(A)	
Write(A)	
Unlock-X(A)	
	Lock-S(A)
	Read(A)
	Unlock-S(A)
	Commit
Rollback	

Hence this must be avoided, and thus basic 2PL fails to resolve these issues.

- Conservative, Strict and Rigorous 2PL

Now that we are familiar with what is Two Phase Locking (2-PL) and the basic rules which should be followed which ensures serializability. Moreover, we came across the problems with 2-PL, Cascading Aborts and Deadlocks. Now, we turn towards the

enhancements made on 2-PL which tries to make the protocol nearly error-free. Briefly, we allow some modifications to 2-PL to improve it. There are three categories:

1. Conservative 2-PL
2. Strict 2-PL
3. Rigorous 2-PL

Now recall the rules followed in Basic 2-PL, over that we make some extra modifications. Let's now see what are the modifications and what drawbacks they solve.

Conservative 2-PL -

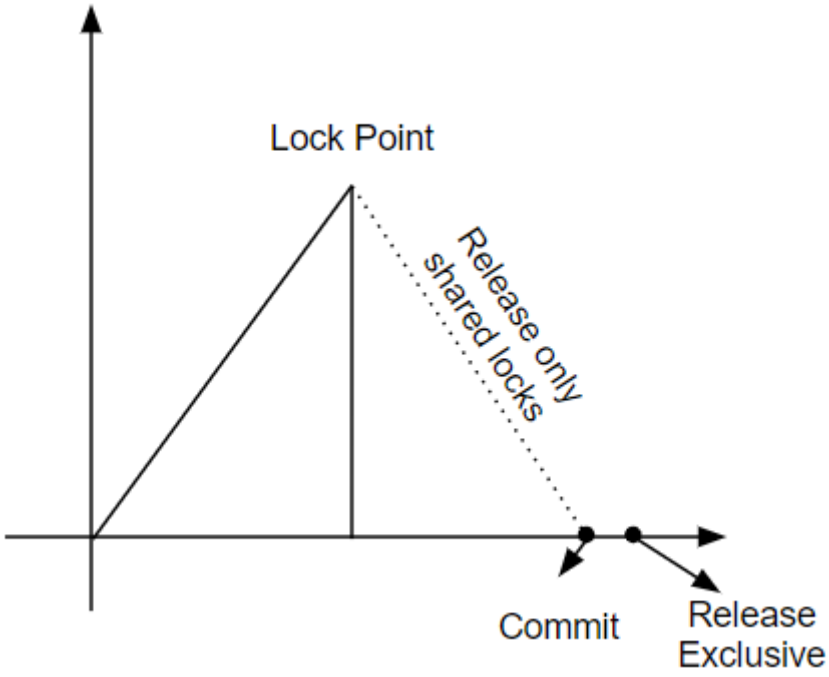
Also known as **Static 2-PL**, this protocol requires the transaction to lock all the items it access before the Transaction begins execution by predeclaring its read-set and write-set. If any of the predeclared items needed cannot be locked, the transaction does not lock any of the items, instead, it waits until all the items are available for locking. Conservative 2-PL is *Deadlock free* and but it does not ensure Strict schedule(More about this here!). However, it is difficult to use in practice because of the need to predeclare the read-set and the write-set which is not possible in many situations. In practice, the most popular variation of 2-PL is Strict 2-PL.

Strict 2-PL -

This requires that in addition to the lock being 2-Phase **all Exclusive(X) Locks** held by the transaction be released until *after* the Transaction Commits.
Following Strict 2-PL ensures that our schedule is:

- Recoverable
- Cascadeless

Hence it gives us freedom from Cascading Abort which was still there in Basic 2-PL and moreover guarantee Strict Schedules but still *Deadlocks are possible*!



Rigorous 2-PL -

This requires that in addition to the lock being 2-Phase **all Exclusive(X) and Shared(S) Locks** held by the transaction be released until *after* the Transaction Commits.
Following Rigorous 2-PL ensures that our schedule is:

- Recoverable
- Cascadeless

Hence it gives us freedom from Cascading Abort which was still there in Basic 2-PL and moreover guarantee Strict Schedules but still *Deadlocks are possible*!

Note the difference between Strict 2-PL and Rigorous 2-PL is that Rigorous is more restrictive, it requires both Exclusive and Shared locks to be held until after the Transaction commits and this is what makes the implementation of Rigorous 2-PL easier.

The Venn Diagram below shows the classification of schedules which are rigorous and strict. The universe represents the schedules which can be serialized as 2-PL. Now as the diagram suggests, and it can also be logically concluded, if a schedule is Rigorous then it is Strict. We can also think in another way, say we put a restriction on a schedule which makes it strict, adding another to the list of restrictions make it Rigorous. Take a moment to again analyze the diagram and you'll definitely get it.

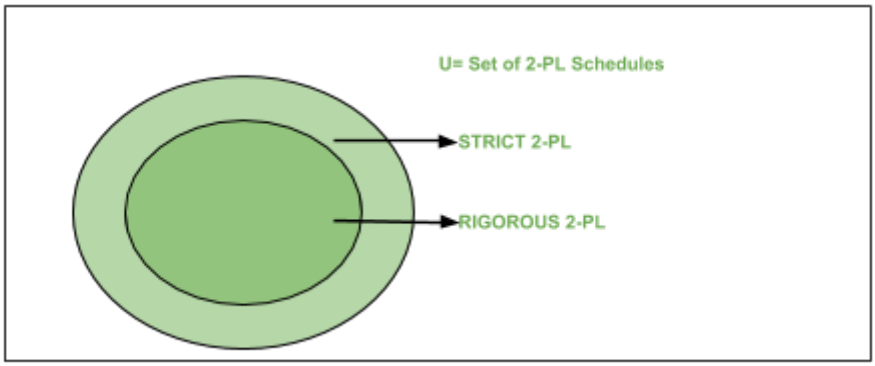


Image - Venn Diagram showing categories of languages under 2-PL

Comparisons:

Conservative 2PL	Strict and Rigorous 2PL
Deadlock Free	Deadlock Possible
Recoverability and Cascadeless-ness not guaranteed	Recoverability and Cascadeless-ness guaranteed

Rigorous Vs Strict:

1. Strict allows more concurrency

2. Rigorous is simple to implement and mostly used protocol

3. Rigorous is suitable in a distributed environment.

- Timestamp based Concurrency Control

Concurrency Control can be implemented in different ways. One way to implement it is by using Locks. Now, lets discuss about Time Stamp Ordering Protocol.

As earlier introduced, **Timestamp** is a unique identifier created by the DBMS to identify a transaction. They are usually assigned in the order in which they are submitted to the system. Refer to the timestamp of a transaction *T* as ***TS(T)***. For basics of Timestamp you may refer here.

Timestamp Ordering Protocol -

The main idea for this protocol is to order the transactions based on their Timestamps. A schedule in which the transactions participate is then serializable and the only *equivalent serial schedule permitted* has the transactions in the order of their Timestamp Values. Stating simply, the schedule is equivalent to the particular *Serial Order* corresponding to the *order of the Transaction timestamps*. Algorithm must ensure that, for each items accessed by *Conflicting Operations* in the schedule, the order in which the item is accessed does not violate the ordering. To ensure this, use two Timestamp Values relating to each database item **X**.

- **W_TS(X)** is the largest timestamp of any transaction that executed **write(X)** successfully.
- **R_TS(X)** is the largest timestamp of any transaction that executed **read(X)** successfully.

Basic Timestamp Ordering -
Every transaction is issued a timestamp based on when it enters the system. Suppose, if an old transaction T_i has timestamp $TS(T_i)$, a new transaction T_j is assigned timestamp $TS(T_j)$ such that **$TS(T_i) < TS(T_j)$**.The protocol manages concurrent execution such that the timestamps determine the serializability order. The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. Whenever some Transaction *T* tries to issue a R_item(X) or a W_item(X), the Basic TO algorithm compares the timestamp of *T* with **R_TS(X) & W_TS(X)** to ensure that the Timestamp order is not violated. This describe the Basic TO protocol in following two cases.

1. Whenever a Transaction *T* issues a **W_item(X)** operation, check the following conditions:
    - If ***R_TS(X) > TS(T)*** or if ***W_TS(X) > TS(T)***, then abort and rollback T and reject the operation. else,
    - Execute W_item(X) operation of T and set W_TS(X) to TS(T).
2. Whenever a Transaction *T* issues a **R_item(X)** operation, check the following conditions:
    - If ***W_TS(X) > TS(T)***, then abort and reject T and reject the operation, else
    - If W_TS(X) <= TS(T), then execute the R_item(X) operation of T and set R_TS(X) to the larger of TS(T) and current R_TS(X).

Whenever the Basic TO algorithm detects twp conflicting operation that occur in incorrect order, it rejects the later of the two operation by aborting the Transaction that issued it. Schedules produced by Basic TO are guaranteed to be *conflict serializable*. Already discussed that using Timestamp, can ensure that our schedule will be *deadlock free*.

One drawback of Basic TO protocol is that it **Cascading Rollback** is still possible. Suppose we have a Transaction T_1 and T_2 has used a value written by T_1. If T_1 is aborted and resubmitted to the system then, T must also be aborted and rolled back. So the problem of Cascading aborts still prevails.

Let's gist the Advantages and Disadvantages of Basic TO protocol:

- Timestamp Ordering protocol ensures serializablity since the precedence graph will be of the form:

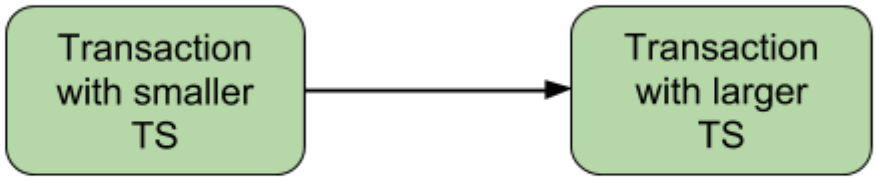


Image - Precedence Graph for TS ordering

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may *not be cascade free*, and may not even be recoverable.

Strict Timestamp Ordering -

A variation of Basic TO is called **Strict TO** ensures that the schedules are both Strict and Conflict Serializable. In this variation, a Transaction T that issues a R_item(X) or W_item(X) such that TS(T) > W_TS(X) has its read or write operation delayed until the Transaction **T'** that wrote the values of X has committed or aborted.

Related GATE Questions -

1. GATE | GATE CS 2010 | Question 20
2. GATE | GATE-CS-2017 (Set 1) | Question 46
3. GATE | GATE-IT-2004 | Question 21

Reference: Database System Concepts, Fifth Edition [Silberschatz, Korth, Sudarshan], Chapter-16

Report An Issue

If you are facing any issue on this page. Please let us know.