4 Transaction and Concurrency Control Techniques

4.1 INTRODUCTION

Definition

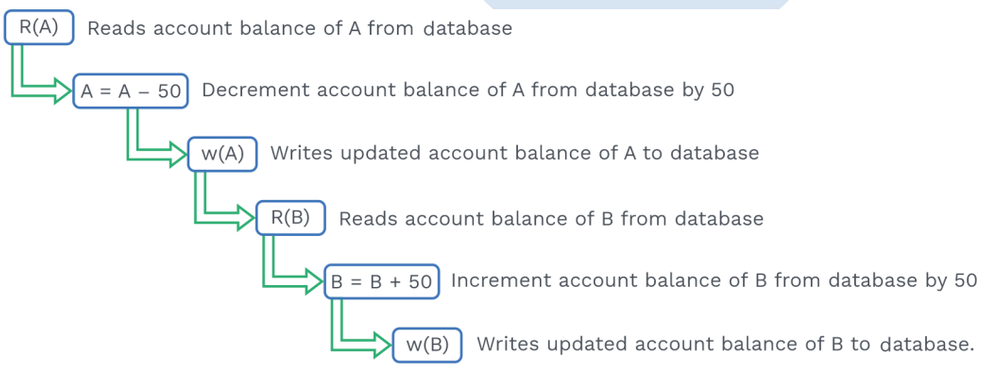
'A transaction is a collection of operations that forms a single logical unit of work'.

For example, if we transfer money from one account to another account, then the transaction consists of two updates one to each account.

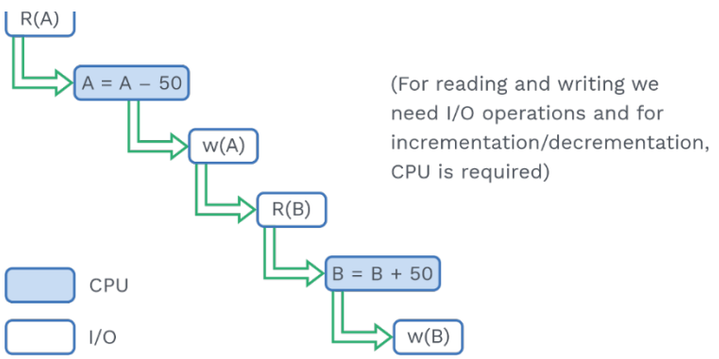
- Consider an example:

Initial account balance of A = 100
Initial account balance of B = 200
Suppose A wishes to transfer ₹ 50 to B.
After transaction account balance of A = 50
After transaction account balance of B = 250

- Let us look at the various steps involved in this single transaction:



- Out of these various steps certain steps require I/O (input-output) operations to be done and certain steps require CPU operations to be done.



- To increase the efficiency of a CPU, it should not remain idle. We can increase the efficiency by assigning CPU to other processes for CPU operations while the CPU is idle.

T_1	I/O	CPU	I/O	I/O	CPU	I/O
T_2	CPU	I/O	CPU	CPU	I/O	CPU

- In this case CPU is not left idle because when transaction T_1 is performing I/O operation, than T_2 is utilising the CPU and when transaction T_2 is performing the I/O operation transaction T_1 is performing CPU operation.

ACID properties

There are 4 ACID properties that a transaction needs to follow:

1)	A	⟶	Atomicity
2)	C	⟶	Consistency
3)	I	⟶	Isolation
4)	D	⟶	Durability

1) Atomicity

Definition

'Atomicity states that either all transactions must reflect properly in the database or none'.

- It means no transaction executes partially.
- Transaction control manager → responsible to ensure atomicity.

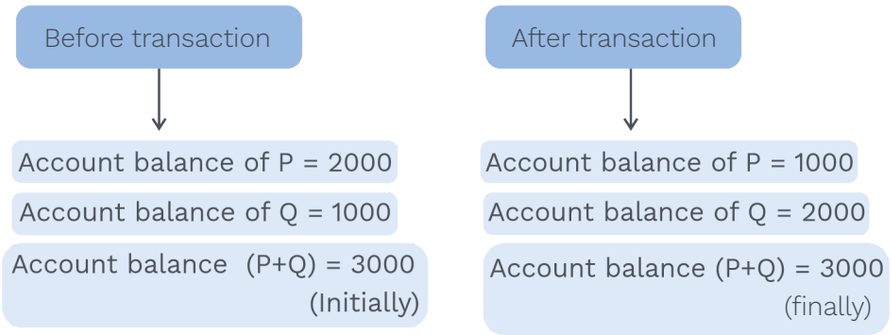
2) Consistency

Definition

'This property ensures that integrity constraints are maintained'.

- In other words, consistency must be there before as well as after the transaction.
- DBMS and application programmer are responsible to ensure consistency of database.

For example, transfer of ₹ 1000 from account P to account Q.



3) Isolation

Definition

'Isolation means each transaction should feel like it is executing alone in the system'.

- The transaction should not feel as if any other transaction is also executing parallelly.

Note:

After executing all the transactions concurrently; the result achieved by the system should be the same as transactions are executing serially (one after the other).

- Concurrency control manager → responsible to ensures isolation.

4) Durability

Definition

This property ensures that if any failure occur (power failure, hard disk crash, etc.) than also system should be able to recover, i.e. even after the failure result of the committed transaction remains same.

- Durability says that whatever changes we are making is permanent, and in future, even there is any failure, these changes will never be lost.

Previous Years' Question

Which of the following is NOT a part of the ACID properties of database transactions?

1) Atomicity **2)** Consistency
3) Isolation **4)** Deadlock-freedom

Sol: 4) **(GATE-2016, Set-01)**

Types of failures

- There are many kinds of failures that can occur in a system.
- Some failures do not result in loss of information but some failures result in loss of information.

1) **Transaction failure:** Logical error and system error are two types of errors that lead a transaction to fail.
   - Example of logical errors are overflow, resource limit exceeded, etc.
   - Deadlock is a system error. As in this case, system enters an undesirable state.
   - Thus, a transaction can't continue its normal execution.

2) **System crash:** Due to the hardware malfunction, a database leads to the loss of content, and therefore transactions might halt.

3) **Disk failure:** Due to the head crash or failure during a data transfer operation, the content of a disk block might be lost.

Transaction states

- A transaction passes through several phases in its life-cycle.

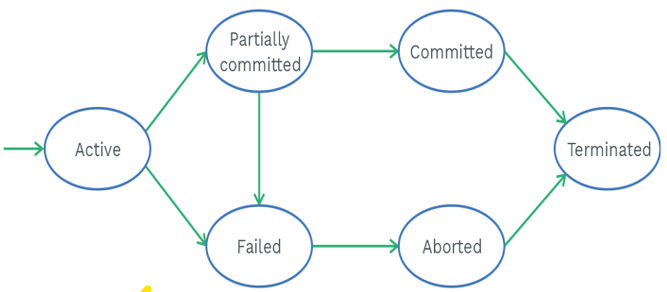


Fig. 4.1 Life Stages of a Transaction

Active state

- It is an initial state of a transaction.

Partially committed state

- A transaction is in a partially committed state when the final operation of a transaction is executed. From here, it is a chance that a transaction might be aborted.

Failed state

- When a transaction discovers that it cannot proceed with its normal execution, it moves to the failed state.

Aborted state

- If a transaction faces roll back, all the changes impacted by the transaction are altered to form the previous image of the database. Consequently, the transaction enters into the aborted life stage.

Committed state

- When a transaction completes its execution successfully, it moves to the committed state.

Note:

Hey learners!!
Do you know what a terminated transaction is?
Well, a terminated transaction is the one that has either been committed or aborted.

- A system has two choices when it moves to the aborted state:
  1) Restart the transaction
  2) Kill the transaction
- Transaction will restart only when the transaction was aborted due to any hardware or software error.

Concurrent executions

- Concurrent execution allows more than one transaction to run concurrently in a transactional system.
- So, inconsistency may be arises when multiple transactions are allowed to update data concurrently.

Why do we need concurrent executions?

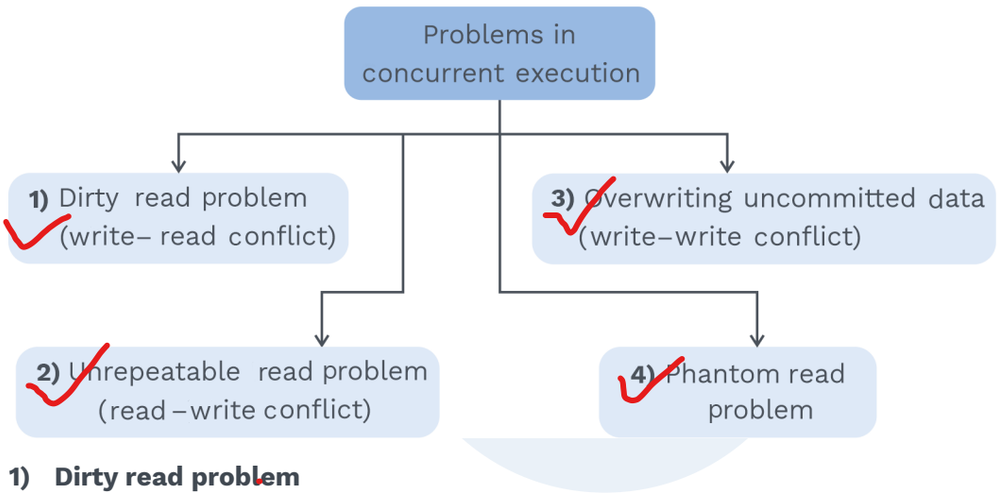
The reasons to allow concurrency are:

1) It enhances database performance and reduces waiting.
2) Better throughput and utilisation of resources can be accomplished.

Problem due to concurrent execution of transactions

- Due to the interleaving of operations between transactions, database can lead to an inconsistent state.

4.2 PROBLEMS IN CONCURRENT EXECUTION



1) Dirty read problem

Definition

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

- Let us make an assumption that a transaction T1 modifies a data item X. Another transaction T2 undergoes read(X). But, due to any reason transaction T1 is failed and the value of the database item is rolled back to the old value.
- So, the value read by T2 is the incorrect one.

Note:

The write-read conflict is also known as reading uncommitted data.

E.g.

	T_1 (A sending ₹ 50 to B)	T_2 (A is incrementing its balance by 4%)

ROLL BACK ↑	R(A) // 100
	A = A – 50 // 50
	w(A) // 50	

		R(A) // 50
		X = A * 0.04
		A = A + X // 52
		w(A) // 52

FAIL	R(B) B = B + 50 w(B)	

When transaction T_1 fails and T_2 executes completely, it will rollback and change the value of A back to its old value.

- So, because transaction T_1 is not completed and committed and T_2 is reading the data written by an uncommitted transaction, T_2 has read the incorrect value of A.

Note:

- Dirty read does not implies inconsistency.
- It only creates problem when the uncommitted transaction fails and rollbacks due to any reason.

2) Unrepeatable read problem (read-write conflicts)

- Assume a scenario where T_1 performs two simultaneous read operations on a specific data item. Another transaction T_2 modifies the content of the data item in the time gap amidst the two read operations.
- Consequently, T_1 will obtain distinct values at different time.

A = 20000 (Initially).

T_1	T_2
R(A) // 20000	
	R(A) // 20000
	A = A – 15000//5000
	w(A)
R(A) // 5000	
A = A – 10000	
w(A)	

- In the above example, when T_1 reads the value of A for the second time, it will read a different value of A because the transaction T_2 has changed the A's value in between first and second read of T_1.
- A real life example of unrepeatable read problems can be considered as:

During a flight reservation, let's say a customer C_1 checks the seat availability and tries to book a ticket meanwhile suppose another customer C_2 booked the ticket so, when customer C_1 reaches to the payment page, it might happen that C_1 will get to read a different value of seat availability.

3) Overwriting uncommitted data (write-write conflicts)

- Write-write conflict problem arises when an update performed by a transaction (say T_1) on a data item is lost due to the update done by another transaction (say T_2).

For example,

Initially A = 100

	T_1 (A Sending ₹ 50 to B)	T_2 (A is incremented its balance by 4%)
	R(A) // 100	
	A = A – 50 // 50	
		R(A) // 100
		X = A * 0.04
		A = A + X // 104
This update is lost ←	w(A) // 50	
		w(A) // 104
	R(B)	
	B = B + 50	
	w(B)	

- This problem is also known as lost-update problem.
- Consider T_1 and T_2 are two transactions executed almost at the same time, and it is already shown in the above figure that operations of these transaction are interleaved.
- So, the final value of item A is not going to be correct because transaction T_2 reads the value of A before it is changed by T_1 in the database.
- This results in loss of the value that is updated by T_1.

$A = 100 \rightarrow A = 50 \rightarrow A = 104$

4) Phantom read problem

- A transaction T_1 reads a set of tuples from a table satisfying the condition given in an SQL query.
- Now, consider another transaction T_2, which inserts a new tuple into the table satisfying the same condition given in the same SQL query present in the transaction T_1.
- T_1 repeats the same query again, then T_1 will result in a row that is not present previously (phantom).
- This situation is known as the phantom read problem.

For example,

T_1	T_2
Eno | Ename | Salary 1 | Asha | 5000 3 | Kimi | 4000 SELECT * FROM Employee WHERE Salary > = 3000;	
	INSERT INTO Employee VALUES (4, Disha, 3500);
Eno | Ename | Salary 1 | Asha | 5000 3 | Kimi | 4000 4 | Disha | 3500 SELECT * FROM Employee WHERE Salary > = 3000;	

(Here we can see SELECT is executed twice, the second time we get additional row.)

Grey Matter Alert!

Hey learners!!
Do you have any idea about the incorrect summary problem?
Now, we will study the incorrect summary problem with an example.

- Suppose one transaction calculating an aggregate summary on different database items.
- While some of these items are updated by other transactions, incorrect summary results can reflect as some values might be calculated by aggregate function before updating their values and some after updating their values.

  For example, let's say we wish to sum up the values of K, X and Y.

  K = 50, X = 100, Y = 200

T_1	T_2	
	Sum = 0	
	R (K)	
	Sum = Sum + K	
R (X)		
X = X + 500		
w (X)		
	R (X)	
	Sum = Sum + X	
	R (Y)	→ T_2 reads X after 500 is added and reads Y before 200 is added, so a wrong summary is a result.
	Sum = Sum + Y	
R (Y)		
Y = Y + 200		
w (Y)		

The sum of K = 50, X = 100, Y = 200 supposed to be 350. But after updating of values of X sum will be (50 + 600 + 200) = 850, if again T_1 tries to increase values of Y by 200 summary will be different.

- Activities like when a transaction starts, ends, aborts or commits must be tracked by the system so that recovery can be possible.
- Following operations are tracked by the recovery manager:

1) Begin transaction
2) Read or write
3) End transaction
4) Commit transaction
5) Rollback (OR) abort

Commit

Commit transaction indicates a successful end of the transaction and whatever updates done by the transaction is committed to the database and later cannot be undone.

Abort (OR) rollback

Abort (OR) rollback indicates that the transaction has ended unsuccessfully. Thus, whatever changes is done by the transaction to the database must be undone.

- The system keeps a log to monitor all the transaction operations so that later it can retrieve all those information in case of any failure.

Commit point of a transaction

- A transaction is said to be reached its commit point if all the operations of a transaction execute successfully and are reflected in the log.

Rack Your Brain

T_1	T_2
R(y)	
	R(x) R(y) y = x + y w(y)
R(y)	

The following given schedule is suffering from:

1) Lost update problem **2)** Unrepeatable read problem
3) Both 1) and 2) **4)** Neither 1) nor 2)

4.3 SCHEDULE

Definition

'When transactions are executing concurrently in an interleaved manner, then the order of execution of operations from any of the various transaction is known as a schedule'.

- We can represent a schedule consisting of n transactions as $T_1, T_2, \ldots, T_n$.
- Interleaving of operations between the transaction is allowed in any schedule.

Note:

Suppose there are two transactions T_1 and T_2 that have to perform m and n number of operations respectively, then total number of possible schedules $= \frac{(m+n)!}{m!\,n!}$

Note:

We will use shorthand notation R, W, C and A for the operation read_item, write_item, commit and abort, respectively, in any schedule.

Serial schedule

Definition

'If the operations of each transaction are executed consecutively, then the schedule is known as serial schedule'.

Problem with serial schedule

- The main problem with serial schedule is that it restricts the interleaving of operations and thus also limits concurrency.
- The serial schedule also leads to the wastage of processing time of the CPU.
- It also starves transactions. For example, if a transaction T_1 is quite long, another transaction has to wait for T_1 to complete its operations.
- Thus, the serial schedule is not good in practice.

Note:

In the serial schedule, transaction should be either T_1 followed by T_2 or T_2 followed by T_1.

Non-serial schedule

Definition

'A schedule S is non-serial, if the operations of each transaction T participating in the schedule, are interleaved or not executed consecutively'.

Note:

- At a time only one transaction is active in serial schedule.
- In serial schedule, next transaction is performed only when an active transaction is committed.
- 'There are n! different valid serial schedules are possible for a set of n transactions'.
- Consistency is always guaranteed if transaction is executing serially.

Rack Your Brain

Find the number of serial schedules possible when three transactions executing?

Example of serial and non-serial schedule

- Let us consider the current values of accounts A and B are ₹ 1500 and ₹ 2500, respectively.
- Consider the following schedule where T_1 is followed T_2:

T_1	T_2
R(A) A = A – 50 w(A) R(B) B = B + 50 w(B)	
	R(A) X = A * 0.2 A = A – X w(A) R(B) B = B + X w(B)

- The final value of accounts A and B, after the execution is ₹ 1160 and ₹ 2840, respectively.
- Hence, total amount of money in accounts A and B remain same after the execution of both transactions.
- Another serial schedule is also possible, i.e. T_2 followed by T_1.

T_1	T_2
	R(A) X = A * 0.2 A = A – X w(A) R(B) B = B + X w(B)
R(A) A = A – 50 w(A) R(B) B = B + 50 w(B)	

- In this case also, the sum A + B is same and the final values in the accounts A and B are ₹ 1150 and ₹ 2850, respectively.
- But consider an another non-serial schedule as shown below.

T_1	T_2
R(A)	
A = A – 50	
	R(A)
	X = A * 0.2
	A = A – X
	w(A)
	R(B)
w(A)	
R(B)	
B = B + 50	
w(B)	
	B = B + X ⟶ This will not preserve consistency
	w(B)

- After the execution of this schedule, we will get final values of accounts A and B are ₹ 1450 and ₹ 2790 respectively.
- Thus, the sum of amount in account A and B is going to increase by ₹ 240 and the final state is inconsistent.

Note:

- It is the responsibility of concurrency control component of a database system to make sure that after the execution of any schedule, the database should remain in consistent state.

Complete Schedule

Definition

'A schedule S of n transactions T_1, T_2, ..., T_n is said to be a complete schedule if the following condition hold:
The last operation of each transaction is either commit or abort operation'.

Example:

T_1	T_2
R (A)	
	R (A)
A = A – 50	
w(A)	
Commit	
	A = A + 50
	w(A)
	Abort

4.4 SERIALIZABILITY

Introduction:

Definition

'A property that tells about the correctness of the schedule when a concurrent transactions are executing'.

Uses of serializability

To maintain the data item in a consistent state.

Serializable schedule

Definition

'A transaction schedule is serializable if its outcome is equal to the outcome of its transactions executed serially'.

Note:

- A schedule that is not equivalent to any one of the serial schedule is known as a non-serial schedule.

Result equivalent schedule

Two schedules are known as result equivalent if the final state of the database comes out to be the same after execution.

Example

Assume initially,

X=2

Y=5

Given below are two schedules. Detect if they are equivalent or not with regard to the results.

T_1	T_2
R(X)//2	
X = X + 5//7	
w(X)	
R(Y)//5	
Y = Y + 5//10	
w(Y)	
	R(X)//7
	X = X * 3//21
	w(X)

Schedule S1

Final value of
X = 21
Y = 10

T_2	T_2
R(X)//2	
X = X + 5//7	
w(X)	
	R(X)//7
	X = X * 3//21
	w(X)
R(Y)//5	
Y = Y + 5//10	
w(Y)	

Schedule S2

Final value of
X = 21
Y = 10

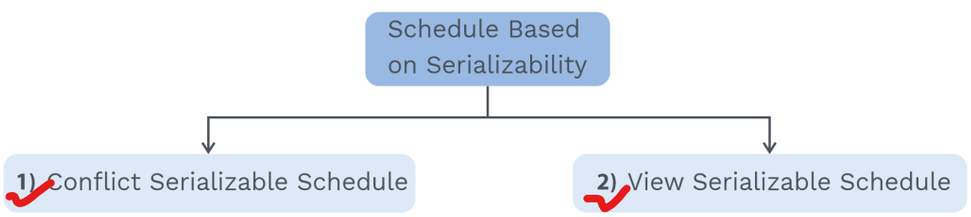
T_1	T_2
	R(X)//2
	X = X * 3//6
	w(X)
R(X)//6	
X = X + 5//11	
w(X)	
R(Y)//5	
Y = Y + 5//10	
w(Y)	

Schedule S3

Final value of
X = 11
Y = 10

Thus schedule S1 which is serial schedule produces the output X = 21 and Y = 10 and non-serial schedule S2 also produces the final output same as S1, i.e.
X = 21 and Y = 10, thus both are result equivalent schedule but final output of S3 is different. It produces output X = 11 and Y = 10, thus S3 is not result equivalent to S1 or S2.

Based on serializability



Conflict serializability

- Let us consider a schedule S where I_i and I_j are the two consecutive operations belongs to transactions T_i and T_j respectively. Here ($i \neq j$).
- Suppose operation I_i and I_j is referring to different data items, then I_i and I_j can be swapped without affecting the outcomes of any instruction in given schedule.

Note:

W(A) denotes write operation on item A.

- When two operations refer to the same data item, order of those operation matters.
- Conflict operations possible are:

	T_i		T_j
	R(A)	.....	W(A)
	W(A)	.....	R(A)
	W(A)	.....	W(A)
Operation	R(A)	.....	R(A) is non-conflicting

Conflict equivalent

Definition

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

- Two operations op_1 and op_2 are conflict operations if
  - **i)** op_1 and op_2 are atomic operations of distinct transactions.
  - **ii)** op_1 and op_2 are operated on the same data item.
  - **iii)** Either of op_1 or op_2 must be a write.
- Conflict equivalence establishes conflict serializability.
- 'A schedule S is known to be conflict serializable if it is conflict equivalent to one of the serial schedule'.

E.g.

T_1	T_2
$R_1(A)$	
$w_1(A)$	
	$R_2(A)$
	$w_2(A)$
$R_1(B)$	
$w_1(B)$	

Schedule S1
(non-serial schedule)

T_1	T_2
$R_1(A)$	
$w_1(A)$	
$R_1(B)$	
$w_1(B)$	
	$R_2(A)$
	$w_2(A)$

Schedule S2
(serial schedule)

Conflicts: $R_1(A) \rightarrow w_2(A)$
$w_1(A) \rightarrow R_2(A)$
$w_1(A) \rightarrow w_2(A)$

Conflicts: $R_1(A) \rightarrow w_2(A)$
$w_1(A) \rightarrow R_2(A)$
$w_1(A) \rightarrow w_2(A)$

Therefore S_1 is conflict equivalent to S_2.

SOLVED EXAMPLES

Q1 **Identify the given schedules are conflict equivalent (or) not?**
S_1: $R_1(A)$ $w_1(A)$ $R_2(A)$ $w_2(A)$ $R_1(B)$ $w_1(B)$ $R_2(B)$ $w_2(B)$
S_2: $R_1(A)$ $w_1(A)$ $R_1(B)$ $w_1(B)$ $R_2(A)$ $w_2(A)$ $R_2(B)$ $w_2(B)$

Sol:

S_1

T_1	T_2
$R_1(A)$	
$w_1(A)$	
	$R_2(A)$
	$w_2(A)$
$R_1(B)$	
$w_1(B)$	
	$R_2(B)$
	$w_2(B)$

S_2

T_1	T_2
$R_1(A)$	
$w_1(A)$	
$R_1(B)$	
$w_1(B)$	
	$R_2(A)$
	$w_2(A)$
	$R_2(B)$
	$w_2(B)$

Conflicts operations in S_1:
$R_1(A) \rightarrow w_2(A)$
$w_1(A) \rightarrow w_2(A)$
$w_1(A) \rightarrow R_2(A)$
$R_1(B) \rightarrow w_2(B)$
$w_1(B) \rightarrow w_2(B)$
$w_1(B) \rightarrow R_2(B)$

Conflicts operations in S_2:
$R_1(A) \rightarrow w_2(A)$
$w_1(A) \rightarrow w_2(A)$
$w_1(A) \rightarrow R_2(A)$
$R_1(B) \rightarrow w_2(B)$
$w_1(B) \rightarrow w_2(B)$
$w_1(B) \rightarrow R_2(B)$

- Here, as we can see all the conflicts are occurring in the same order in both schedules are same.
- Thus, schedule S_1 which is a non-serial schedule is conflict equivalent to serial schedule S_2.
- Therefore, schedule S_1 is a conflict serializable schedule.

Consider the transactions T_1, T_2 and T_3 and the schedules S_1 and S_2 given below:

Previous Years' Question

T_1: $r_1(x)$; $r_1(z)$; $w_1(x)$; $w_1(z)$
T_2: $r_2(x)$; $r_2(z)$; $w_2(z)$
T_3: $r_3(x)$; $r_3(x)$; $w_3(y)$
S_1: $r_1(x)$; $r_3(y)$; $r_3(x)$; $r_2(y)$; $r_2(z)$; $w_3(y)$; $w_2(z)$; $r_1(z)$; $w_1(x)$; $w_1(z)$
S_2: $r_1(x)$; $r_3(y)$; $r_2(y)$; $r_3(x)$; $r_1(z)$; $r_2(z)$; $w_3(y)$; $w_1(x)$; $w_2(z)$; $w_1(z)$
Which of the following statements about the schedules is TRUE?

1) Only S_1 is conflict-serializable
2) Only S_2 is conflict-serializable
3) Both S_1 and S_2 are conflict serializable
4) Neither S_1 nor S_2 is conflict serializable

Sol: 1) **(GATE-2014, Set-03)**

How to test conflict serializability for a given schedule

- Precedence graph helps to ascertain if a particular schedule is conflict serializable or not.
- A precedence graph is based on read and write operations.

Note:

Hey learners!!
Do you have any idea about the precedence graph (Serialization graph)?
Let us read the definition of the precedence graph.

- Definition: 'Precedence graph (Serialization graph) is a directed graph G = (V, E) that consists of a set of nodes V = {T_1, T_2, ..., T_n} and a set of directed edges E = {e_1, e_2, ..., e_m}'.

Algorithm

Following are the steps to draw serialization (precedence) graph:

1)	For all the transactions present in schedule S, create a node as T_i.
2)	Make an edge ($T_i \longrightarrow T_j$) if there is any conflict operation such as write–read (OR) read–write (OR) write–write from transactions T_i to T_j.

- If there is no cycle present in the precedence graph, it means the schedule is conflict serializable.

Note:

Hey Learners!!

The schedule S may or may not be conflict serializable if cycle is present in the precedence graph.

Grey Matter Alert!

$(T_i \rightarrow T_k)$, $(T_k \rightarrow T_p)$, $(T_p \rightarrow T_j)$, $(T_j \rightarrow T_i)$ is a cycle in a directed graph.

Note:

- Topological sorting is used to obtain the serializability order of any transaction.
- We can obtain more than one possible linear order using topological sorting.

Example: Identify if the given non-serial schedule S is conflict serializable?

T_1	T_2
R(A) w(A)	
	R(A) w(A)
R(B) w(B)	
	R(B) w(B)

Schedule S

Sol: We will check whether this non-serial schedule is conflict serializable or not using precedence graph.

I) Transactions T_1 and T_2 represents node of precedence graph.

II) Conflict operations are represented by edges.
The conflicting operations are:
$R_1(A) \rightarrow W_2(A)$
$W_1(A) \rightarrow R_2(A)$
$W_1(A) \rightarrow W_2(A)$
$R_1(B) \rightarrow W_2(B)$
$W_1(B) \rightarrow W_2(B)$
Which are from transactions T_1 to T_2. There are no conflicts from T_2 to T_1.

So, serialization graph has no cycle. Thus, the given non-serial schedule S is conflict serializable.

SOLVED EXAMPLES

Q2 **Identify the following schedule is conflict serializable or not?**

T_1	T_2	T_3
R(X)		
		R(X)
W(X)		
	R(X)	
	W(X)	

Sol: For the provided schedule, the precedence graph is computed. It will determine if that schedule is conflict serializable or not.

1) Transactions T_1, T_2 and T_3 represent the node of the graph.

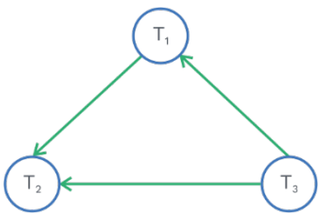
2) Conflict operations:

$R_1(X) \rightarrow W_2(X)$
$W_1(X) \rightarrow R_2(X)$ } Edge from T_1 to T_2
$W_1(X) \rightarrow W_2(X)$

$R_3(X) \rightarrow W_2(X) \longrightarrow$ Edge from T_3 to T_2

$R_3(X) \rightarrow W_1(X) \longrightarrow$ Edge from T_3 to T_1

So, the precedence graph will be:



There is non-existence of a cycle in the above serialization graph, we obtained. So, the given schedule is conflict serializable. Thus, the possible serial schedule that follows from the topological sorting of the obtained serialization graph:
$T_3 \rightarrow T_1 \rightarrow T_2$

Q3 **Test whether the following schedule is conflict serializable or not.**

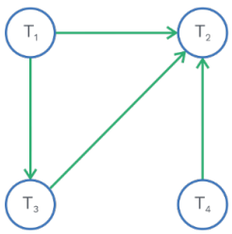
T_1	T_2	T_3	T_4
			R(A)
	R(A)		
		R(A)	
W(B)			
	W(A)		
		R(B)	
	W(B)		

Sol: We will first check using precedence graph whether the given schedule is conflict serializable or not.

1) T_1, T_2, T_3, T_4 represents node of the graph.

2) Conflict operations are represented by edges:
$W_1(B) \rightarrow W_2(B)$: Edges from T_1 to T_2
$R_3(A) \rightarrow W_2(A)$: Edges from T_3 to T_2
$R_4(A) \rightarrow W_2(A)$: Edges from T_4 to T_2
$R_3(B) \rightarrow W_2(B)$: Edges from T_3 to T_2
$W_1(B) \rightarrow R_3(B)$: Edges from T_1 to T_3

So, we will get precedence graph as:



The above graph contains no cycle. Therefore, Conflict serializable.
Possible Serialized schedule after applying topological sort:

1. $T_1 \rightarrow T_3 \rightarrow T_4 \rightarrow T_2$ 2. $T_1 \rightarrow T_4 \rightarrow T_3 \rightarrow T_2$ 3. $T_4 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2$

Previous Years' Question

Let $r_i(z)$ and $w_i(z)$ denote read and write operations, respectively, on a data item z by a transaction T_i.
Consider the following two schedules:
S_1: $r_1(x)$ $r_1(y)$ $r_2(x)$ $r_2(y)$ $w_2(y)$ $w_1(x)$
S_2: $r_1(x)$ $r_2(x)$ $r_2(y)$ $w_2(y)$ $r_1(y)$ $w_1(x)$
Which one of the following options is correct?

1) S_1 is conflict serializable and S_2 is not conflict serializable.
2) S_1 is not conflict serializable and S_2 is conflict serializable.
3) Both S_1 and S_2 are conflict serializable.
4) Neither S_1 nor S_2 is conflict serializable.

Sol: 2) **(GATE-CSE-2021, Set-01)**

View serializability

Consider two schedules S1 and S2 having the same set of operation.

1)	If T_i reads initial value of A in S_1, then T_i should also read initial value of data item A in S_2.
2)	If T_J performs final write operation on data item A in schedule S_1 then T_j should also perform the final write operation on A in schedule S_2.
3)	If T_j reads the value produced by T_i in schedule S_1 then T_j must also read the value produced by T_i schedule S_2.

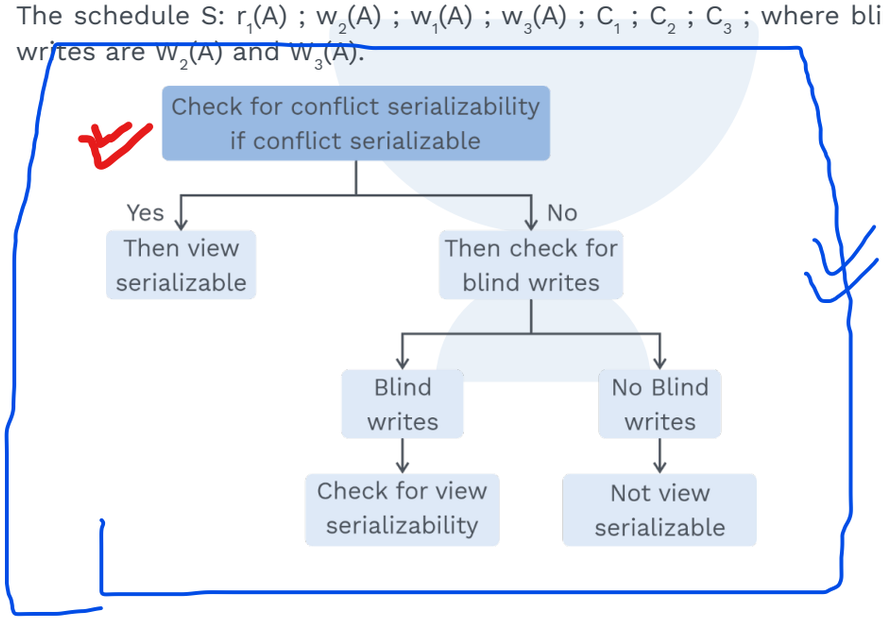
Note:

When a transaction performs a write operation on any data item (say A) without any prior read. It is known as blind write.

Note:

The condition for a schedule to achieve view serializability is: It should be viewed equivalent to either of the realizable serial schedules.

E.g. Consider three transactions: T_1: $r_1(A)$; $w_1(A)$; T_2: $w_2(A)$; and T_3: $w_3(A)$; The schedule S: $r_1(A)$; $w_2(A)$; $w_1(A)$; $w_3(A)$; C_1 ; C_2 ; C_3 ; where blind writes are $W_2(A)$ and $W_3(A)$.



Note:

i) View serializability is a necessary condition for a schedule to be conflict serializable but opposite is not true.

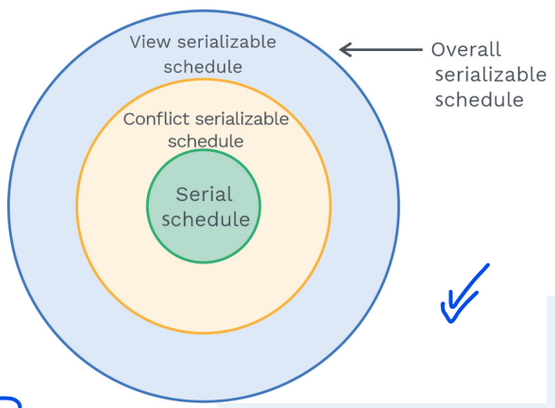


Fig. 4.2 Diagrammatic Representation of Conflict Serializable and View Serializable Schedule

Example: Determine if the following schedules are view equivalent or not.

S_1

T_1	T_2
R(A)	
A = A + 10	
W(A)	
R(B)	
B = B + 20	
W(B)	
	R(A)
	A = A + 10
	W(A)
	R(B)
	B = B * 1.1
	W(B)

S_2

T_1	T_2
R(A)	
A = A + 10	
W(A)	
	R(A)
	A = A + 10
	W(A)
R(B)	
B = B + 20	
W(B)	
	R(B)
	B = B * 1.1
	W(B)

i) If T_1 undergoes initial read operation on any data item X in S_1, it should definitely read the initial value of X in S_2 also.

ii) $W_1(A) \rightarrow R_2(A)$
$W_1(B) \rightarrow R_2(B)$ holds for both schedule.

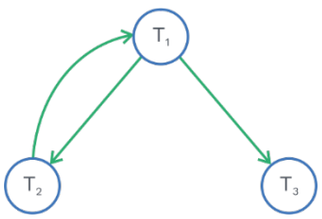
iii) In either of the schedules, final write on data item A and B is performed by T_2. Thus, schedule S_2 is view serializable to S_1.

SOLVED EXAMPLES

Q1 **Check whether the given schedule is view serializable or not.**

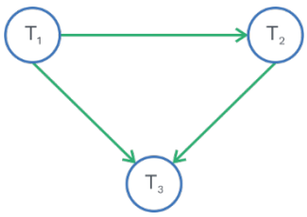
T_1	T_2	T_3
R(A)		
	W(A)	
W(A)		
		W(A)

Sol: To find out whether a given schedule is view serializable or not, first, we will find out whether a given schedule is conflict serializable or not, because we know if a schedule is conflict serializable then schedule is also view serializable.
Conflict serializability check:



- The above graph contains cycle, thus it is not conflict serializable.
- Check if there is any blind write, if no blind write then it is not view serializable schedule.
- In the given schedule blind write is present.
- So, now we will test for view serializability for that we will draw polygraph.
- To draw polygraph, following steps are required:
  i) T_1, T_2, T_3 represents nodes of the graph.
  ii) T_1 will execute first, as T_2 is depending on T_1 and T_3 is depending on T_1.

- T_3 should be the last one to execute, based on the writes.
- We will get the following graph:



So, the serial schedule we will get is: $T_1 \rightarrow T_2 \rightarrow T_3$.
(Topological order from topological sort.)
Thus the given schedule is view serializable.

Q2 **Consider the following given schedule:**

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

Is it view serializable schedule?

Sol: First, we will find out whether a given schedule is conflict serializable or not. For that we will draw precedence graph.

- A cycle is present in the above graph. So, it is not conflict serializable.
- Check if there is any blind write, if no blind write then it is not view serializable schedule.
- In the given schedule blind write is present.
- Now, we will test for view serializable, so we will draw polygraph.

T_1 has to start first as read operation is performed by T_1 and since final write operation is also performed by T_1. So, there will be an edge from T_2 to T_1. It means graph is having cycle.
$\therefore$ We can conclude that the given schedule is neither conflict serializable nor view serializable.

Q3 **Is the following given schedule view serializable?**
T_1: R(x), T_2: R(y), T_1: W(x), T_2: R(y), T_3: W(y), T_1: W(x), T_2: R(y)

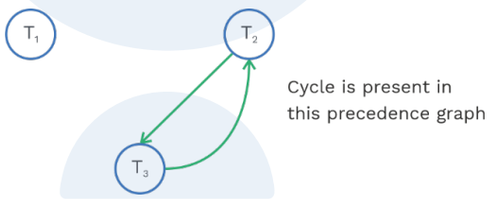
Sol:

T_1	T_2	T_3
R(x)		
	R(y)	
W(x)		
	R(y)	
		W(y)
W(x)		
	R(y)	

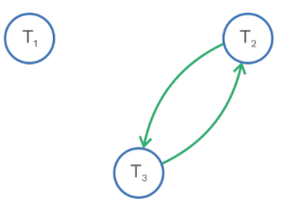
First, we will draw a precedence graph to check conflict serializable schedule:

I) T_1, T_2, T_3 represents nodes of a graph.

II) All conflict operations represent edges of precedence graph.



- Thus, the given schedule is not a conflict serializable schedule.
- We will check if there is any blind write, if no blind write then it is not view serializable schedule.
- In the given schedule blind write is present.
- Now, draw polygraph to check view serializability



As T_1 is the only transaction that performs read and write on variable x so the initial we are ignoring T_1.

T_2 has to start first as initial read operation is performed by T_2 on variable y, there will be an edge from T_2 to T_3.

Due to updated read dependency from T_3 to T_2, there will be an edge from T_3 to T_2.

So, there is cycle in the graph. So, the given schedule is not view serializable.

Rack Your Brain

How many conflicting operations does schedule S have _____?
Schedule S: $r_1(M)$ $w_2(M)$ $r_2(M)$ $w_1(N)$ $r_2(N)$ $w_2(N)$

4.5 BASED ON RECOVERABILITY

- To ensure the atomicity of the transaction, we undo the effect of the transaction whenever it fails.
- Also if a transaction T_j is depending on T_i that needs to be aborted.
- We need to put some restrictions on the schedule to achieve this.

Irrecoverable schedule

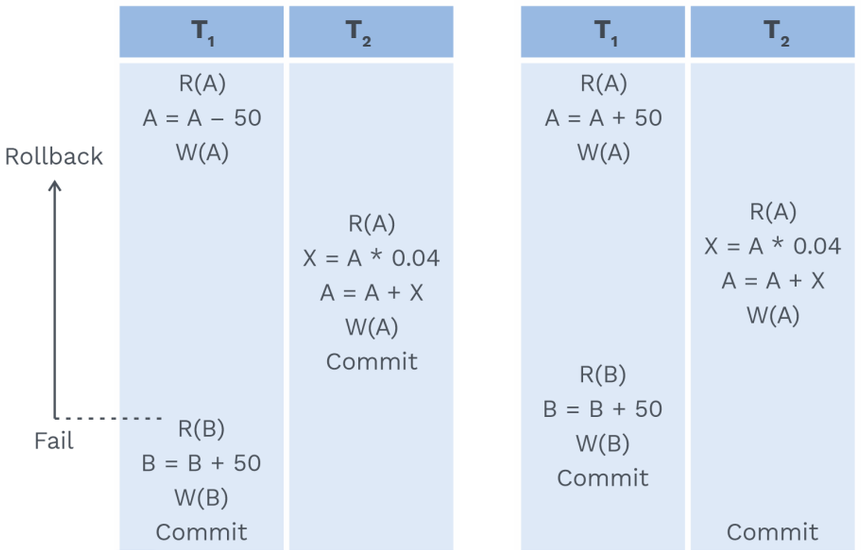
T_1	T_2
R(A)	
W(A)	
	R(A)
	Commit
R(B)	

- The above given schedule is not recoverable (irrecoverable) schedule since commit(T_2) happens before commit(T_1).
- Now suppose T_1 fails before it commits. T_2 has to undergo abort.
- But abort(T_2) is not possible as it has already undergone commit operation.
- So, it is a situation where it is not possible to recover from the failure of T_1.

Recoverable schedules

'A schedule where for each pair of transactions T_i and T_j the commit operation of T_i must appear before the commit operation of T_j if T_j reads a data item previously written by T_i'.

- Given below is an example depicting instances of recoverable and irrecoverable schedules:



Note:

Hey Learners!!
Do you know what a recoverable schedule guarantees?
Well, it guarantees that a transaction does not need to roll back once it commits.

Ensuring recoverability of schedule

T_2 should commit only after T_1 commits if T_2 depends on T_1.
If the above condition is satisfied then it is guaranteed to have a recoverable schedule.

Example: The below given schedule is an example of recoverable schedule.

T_1	T_2
W(A)	
	R(A)
Commit	
	Commit

Note:

A recoverable schedule may have dirty read problem.

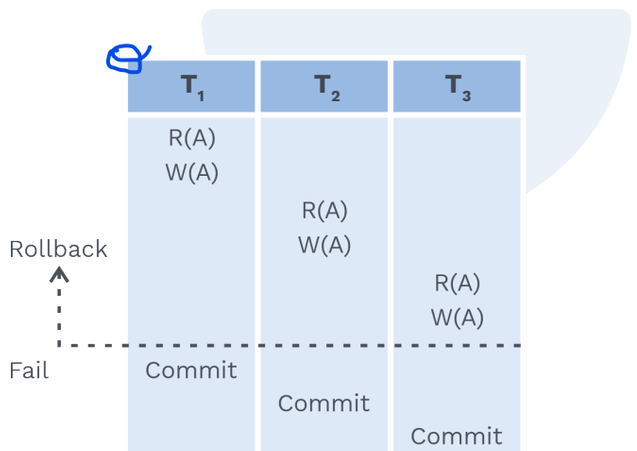
Note:

Set of the recoverable schedules are subset of set of all possible schedules.

Cascading Aborts

If one transaction failure causes multiple transactions to roll back, it is called as cascading roll back (or) cascading aborts.

Example:



- Here, rollback of T_1 will result in rollback of T_2 and rollback of T_2 will result in rollback of T_3 which is cascading rollbacks.
- It is recoverable schedule but due to cascading rollbacks, other problem arises like less throughput, more waiting time.
- Just because of rollback of T_1 here T_2, T_3, T_4 all needs to rollback.
- Thus, cascading rollbacks must be avoided.

Cascadeless schedule

Cascading rollback is not desirable as they undo a good amount of work. So, it is better if we can avoid the cascading rollback.

Schedule that is restricted in such a way that cascading rollback cannot occur is known as cascadeless schedules.

Definition

'A cascadeless schedule is one, where for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j'.

Note:

Hey learners!!
Do you know that a cascadeless schedule is a recoverable one?
See the below example.

Example:

T_1	T_2	T_3
R(A) W(A) Commit		
	R(A) W(A) Commit	
		R(A) W(A) Commit

→ Cascadeless schedule

Strict schedule

'Strict schedules are those where a value written by a transaction cannot be read or written by another transaction until the previous transaction commits (or) aborts'.

Example:

S :

T_1	T_2
R(A) W(A) Commit	
	W(A)/R(A)

Here, S is a strict schedule, i.e. it is both cascadeless as well as recoverable.

Note:

Set of all cascadeless schedule are subset of set of recoverable schedule.

Note:

Set of all strict schedule are subset of set of all cascadeless schedule.

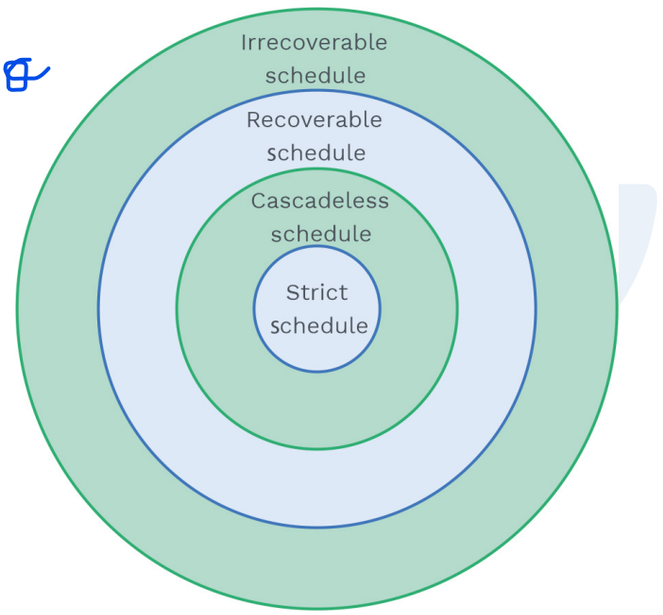
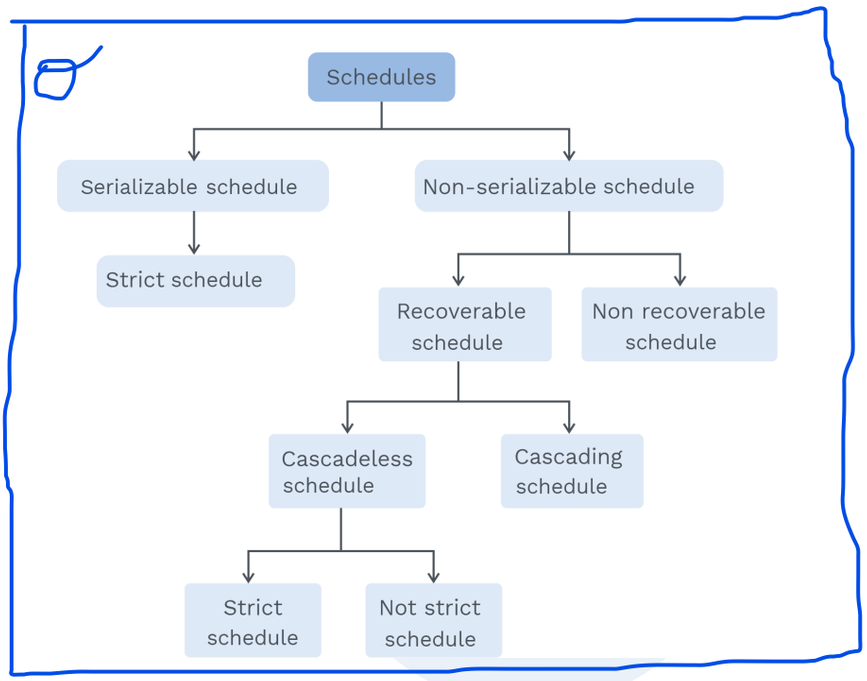
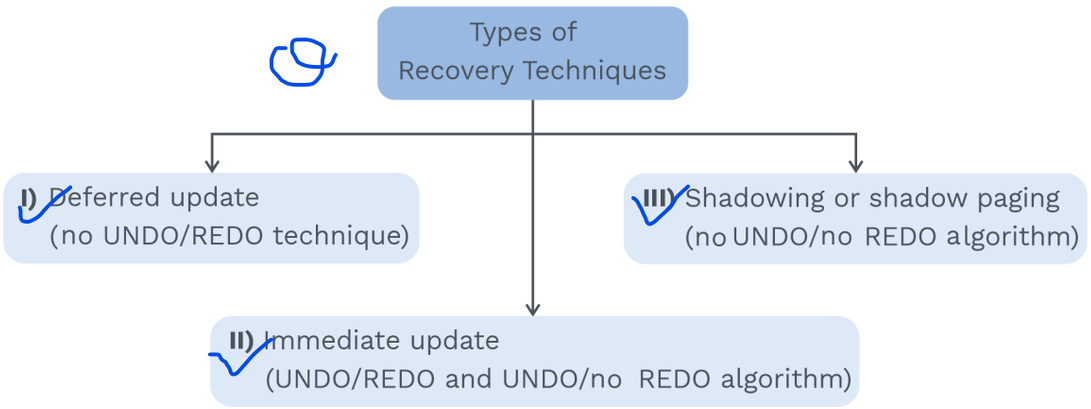


Fig. 4.3 Different Type of Schedules Based on Recoverability



Database recovery techniques

- Failures can occur due to system crashes or transaction errors, etc.
- There are few techniques that we can use to recover from these failures.
- The recovery process uses commit point, system log concepts to recover from any failure.



Checkpointing in the system log

- 'Checkpointing is a type of entry in the log'.
- Generally a record is written into the log periodically and all the updates are recorded on the disk during checkpointing.
- So, transactions that are committed in the log before checkpointing needs not to execute their WRITE operation if any system crash occurs.

- Usually, log records are denoted as:
- $<T_i$, start> × Transaction T_i has started.
- $<T_i$ × X_j, V_1, V_2> × Transaction T_i has performed a write on data item X_j.

Before the write, X_j had value V_1 and after the write X_j will have value V_2.

- $<T_i$, commit>, Transaction T_i has committed.
- $<T_i$, abort>, Transaction T_i has aborted.

Why we need checkpoints?

1) It is very time consuming to search the entire log in case of any failure, thus it is always a good idea to maintain checkpoints in the log.

2) Checkpoint also increase the throughput of the system as we need not to waste time in a recovery of a transaction in case if we need to execute that transaction again due to any reason.

UNDO/REDO recovery algorithm with checkpoints

1) There are two list of transactions that needs to be maintained by the system:

i) The active transactions

ii) Committed transactions since the last checkpoint.

2) Perform UNDO for every write operation of the uncommitted/active transaction.

3) Perform REDO for every write operation of a committed transaction.

Example: Consider the given checkpointing protocol and the operations given in the log.

(start, T_1)
(write, T_1, y, 3, 2)
(start, T_2)
(commit, T_1)
(write, T_2, z, 4, 7)
(checkpoint)
(start, T_3)
(write, T_3, x, 1, 8)
(commit, T_3)
(start, T_4)
(write, T_4, z, 7, 2)

Now, if crash occurs,the system will try to recover. The undo and redo operations are as follows:
Undo list: T_2, T_4 (It means transaction T_2 and T_4 will be undone.)
Redo list: T_3 (It means transaction T_3 will be redone.)

4.6 CONCURRENCY CONTROL TECHNIQUE

- We know that isolation is one of the important properties that a transaction needs to follow.
- It is mandate while concurrent implementation of transactions to preserve isolation property.
- A concurrency control technique is used to achieve this.

Note:

Concurrency control protocols guarantee serializability.

4.7 LOCK-BASED PROTOCOLS

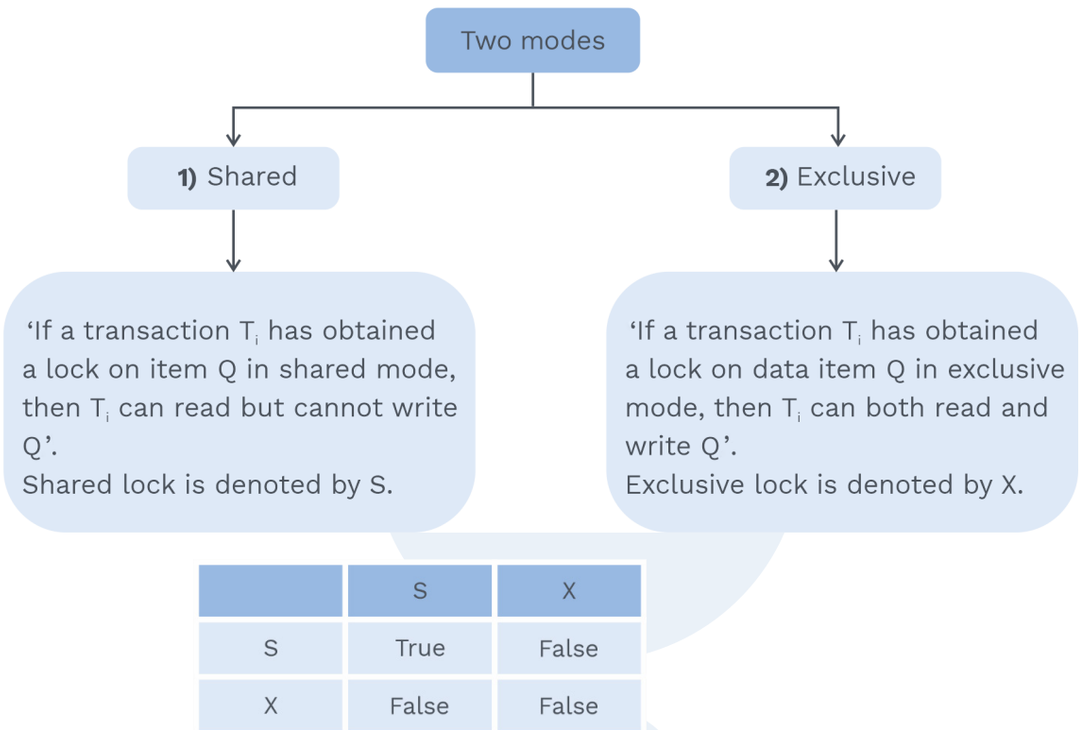
- If we follow mutual exclusion to access the data items, then we can ensure the serializability.
- It means any other transaction cannot modify the data items while the same data item is being accessed by some other transaction.
- We use lock based protocols for this purpose.
- A transaction can have access to a data item if any lock on that specific item is currently held by that transaction.

Definition

'A lock is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it'.

Lock

- There prevails two distinct locking modes: shared and exclusive.



	S	X
S	True	False
X	False	False

Fig. 4.4 Relationship Between Shared and Exclusive Lock

- A transaction acquires a lock in a particular mode based on the types of operation it wants to perform on the data item Q.
- The concurrency control manager is going to grant these locks to a transaction.
- After getting the locks, a transaction can proceed.

Note:

- Figure 4.4 represents that a shared mode is compatible with shared mode but it is not compatible with exclusive mode.
- The instruction lock-S(A) imparts shared lock on data item, A.
- Similarly, To get a exclusive lock on data item A, we have to use lock-X(A) instruction.
- To unlock a data item A, we have to use unlock(A).
- Locking a data item by a transaction is mandate to have access on it.
- If a transaction has acquired an incompatible lock on a data item, no other lock can be provided to another transaction to access the same data item until that lock has been released. In this case, another transaction has to wait.

Difference between shared and exclusive lock

Shared lock imparts read access on a specific data item. However, exclusive lock mode imparts both read and write grants to a transaction on a targeted data item.

Example: Folks!!

Consider there are two accounts called A and B.
Transactions T_1 and T_2 can access these two accounts.
Suppose transaction T_1 transfers rs. 50 to account A from account B.
Total amount (A+B) is displayed by transaction T_2.

T_1 :	T_2 :
L_X (B);	L_S (A);
R (B);	R (A);
B = B – 50;	U (A);
W (B);	L_S (B);
U (B);	R (B);
L_X (A);	U (B);
R (A);	display (A + B);
A = A + 50;	
W (A);	
U (A);	

Note:

In the above example, L represents lock, R represents read_item, W represents write_item and U represents unlock. We will consider these notation further also.

- Let the value of A and B is ₹ 100 and ₹ 200 initially. If we execute T_1, T_2 serially then T_2 will display A + B = ₹ 300.
- But if these transactions are executed concurrently, then there may be the case possible such that T_2 will display incorrect result.

Example:

T_1	T_2
L_X(B)	
R(B)	
B = B – 50	
W(B)	
U(B)	
	L_S(A)
	read(A)
	U(A)
	L_S(B)
	read(B)
	U(B)
	display (A + B)
L_X(A)	
read (A)	
A = A + 50	
Write (A)	
U(A)	

→ Produces inconsistent result.

- Releasing the lock too early in the simple locking case will lead to inconsistent result.
  (T_2 produces an inconsistent result as T_1 is unlocking the data items too early.)
- Simple locking can also lead to an undesirable situation.

Example:

T_1	T_2
L_X(B)	
read(B)	
B = B – 50	
Write(B)	
	L_S(A)
	read(A)
	L_S(B)
L_X(A)	

Here, the above example is leading to deadlock as T_2 wants a shared lock on B but T_1 holds an exclusive lock on B.

Similarly, T_1 wants an exclusive lock on data item A that is held by T_2 in shared mode. Thus, T_1 is waiting for T_2 to unlock A and T_2 is waiting for T_1 to unlock B.

Grey Matter Alert!

Hey Learners!!
Do you know what happens when a deadlock is present?
When none of the transactions is able to proceed with its normal execution. This situation is known as deadlock.

Note:

If deadlock persists, either of the transactions must undergo roll-back/abort by the system.

Drawbacks of simple locking

i) It leads to an inconsistent result.
ii) Simple locking can also lead to deadlock.
iii) Simple locking does not guarantee serializability.

Example of simple locking schedule that does not guarantee serializability is as follows:

T_1	T_2
lock_X(A)	
write(A)	
Unlock_X(A)	
	lock_S(A)
	read(A)
	Unlock_S(A)
lock_X(A)	
write(A)	
Unlock_X(A)	

This is not conflict serializable schedule as to when we will draw precedence graph, cycle will be present.

Conflict operations are:

1) $W_1(A) \rightarrow R_2(A)$

2) $R_2(A) \rightarrow W_1(A)$

How to grant locks?

- When a transaction asks for a lock on a data item in a particular mode and there is no other transaction that holds a lock on the same data item in a conflicting mode, then a lock can be granted.
- Starvation is a situation where a particular transaction does not make progress as every time a lock is granted to other transaction.

Two phase locking protocol

- Two phase locking protocol always guarantee serializability.
- In 2PL, locking and unlocking is done in 2 phases:
  1) **Growing phase:** 'A transaction can acquire new locks on items but no other lock can be released'. It is also known as expanding phase.
  2) **Shrinking phase:** 'During this phase a transaction can release existing lock but cannot acquire new locks'.

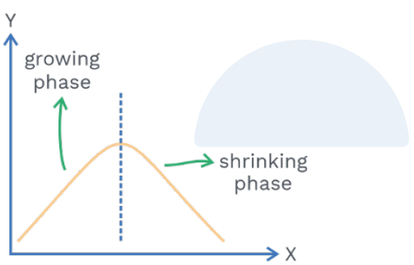


Fig. 4.5 Graphical Representation of 2PL

Note:

Initially, a transaction obtains locks, and it is said to be in growing phase.
After that, a transaction enters the shrinking phase.
Once a transaction initiates lock release phenomenon, it cannot generate any further lock acquisition requests.

T_1	T_2	T_3
	L-S(A)	L-S(A)
	L-X(B)	
	U(A)	L-X(C)
	U(B)	
L-S(B)		
		U(A)
		U(C)
L-X(A)		
U(A)		
U(B)		

(After taking/obtaining all the locks only, unlocking begins)

Note:

If two phase locking protocol is followed by all the transactions of a schedule then it is definitely a serializable schedule.

Note:

Same as basic 2PL, 2PL with lock conversion gives only conflict serializable schedule.

Grey Matter Alert!

Lock upgradation	Lock downgradation
To upgrade a shared lock to an exclusive lock.	To downgrade an exclusive lock to a shared lock.

- Upgradation is allowed only in the growing phase whereas down gradation is allowed only in the shrinking phase.
- In the following example, W(A) and W(B) will not be allowed since T_i has exclusive lock.

T_i	T_j
L-X(A) W(A)	
	W(A) W(B) → Blocked
L-X(B) W(B)	

So, how can we determine the order in which serializability is obtained? For that, there is something called a lock point.

Lock point

Definition

Point at which a transaction gets its final lock.
Lock point is used to determine the order of the transactions.

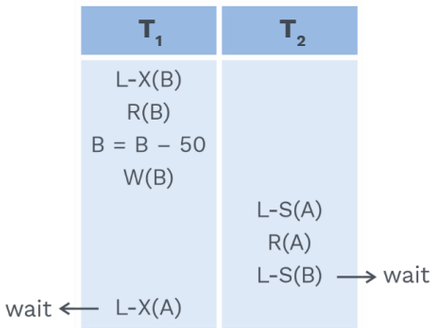
Drawbacks of two-phase locking protocol

1) 2PL may leads to cascading rollback.
2) Deadlock might be possible.
3) Starvation is also possible in 2PL.
Example: Consider a schedule S such that

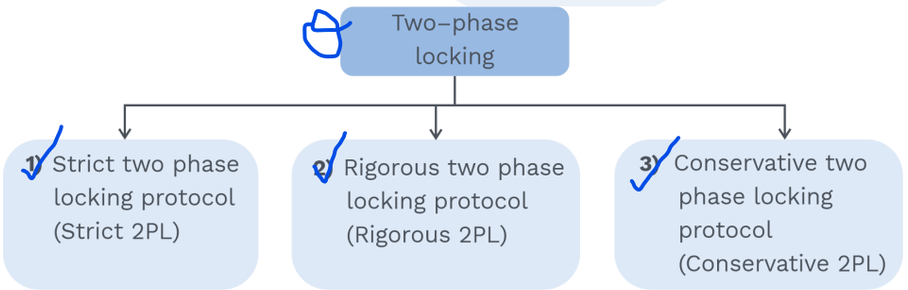
T_1	T_2	
L_X(A)		
W(A)		
L_X(B)		
W(B) ⋮		
U(A)		
U(B)		
	L_S(A)	If T_1 rollback then T_2 also need to rollback
	R(A)	

Rollback ↑

- Example detecting deadlock.



- Here, T_1 is having exclusive lock on data item B and is requesting exclusive lock on data item A.
- Similarly, T_2 is having shared lock on data item A and requesting shared lock on data item B, which clearly says that deadlock is present.
- Which clearly says that deadlock is present.
- There are three types of two-phase locking.



Strict 2PL

- Strict 2PL guarantees strict schedules.

Definition

'It is a variation of 2PL, a transaction T does not release any of its exclusive (read and write) locks until it commits or aborts'.

- So, any other transaction cannot read or write an item written by T until T has committed.
- It leads to a recoverable schedule.
- Strict 2PL ensure that a schedule is:
  1) Recoverable schedule **2)** Cascadeless schedule

- The advantage of strict 2PL is that it avoids cascading rollbacks.
- Strict 2PL is not deadlock free.

Example:

T_1	T_2
L-X(A) W(A) Commit U(A)	
	L-S(A) R(A) U(A) Commit

(Since lock on A is already given to T_1, it will never be granted to T_2, that is T_2 will not able to read/write same data item until transaction T_1 that has written the data item performs commit.)

Note:

Hey Learners!!
Do you know how can we decide the order of transactions in the strict 2PL?
Well, the order of transactions is decided by the sequence of lock points.

Whatever final order, we get is equivalent to serial schedule.

Rack Your Brain

Strict 2PL protocol guarantees:
1) Recoverable schedule
2) Cascadeless schedule
3) Strict schedule
4) All of the above

Previous Years' Question

Consider the following database schedule with two transactions T_1 and T_2.
$S = r_2(x); r_1(x); r_2(y); w_1(x); r_1(y); w_2(x); a_1 ; a_2$
Where $r_i(z)$ denotes a read operation by T_i on a variable z and a_i denotes an abort by transaction T_i.
Which one of the following statements about the above schedule is TRUE?

1) S is non-recoverable
2) S is recoverable, but has a cascading abort
3) S does not have a cascading abort
4) S is strict

Sol: 3). **(GATE-2016, Set-2)**

Rigorous 2PL

- Rigorous 2PL is more restrictive variation of strict 2PL.

Definition

'In rigorous 2PL, in addition to locking being in 2 phase, a transaction T does not release any of its locks (shared or exclusive) until it commits or aborts'.

- Rigorous 2PL also guarantees strict schedules.

Conservative 2PL

Definition

In conservative 2PL, all the locks acquired will not be released until transaction commits.

- It says that once the transaction begins it is in its shrinking phase and then transaction is in its expanding phase until it finishes.

- Advantages of conservative 2PL:
  1) Conservative 2PL gives strict schedules.
  2) It also gives freedom from deadlock.

Note:

Conservative 2PL may lead to starvation.

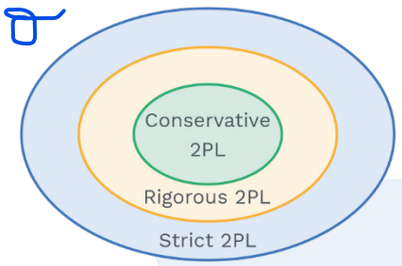


Fig. 4.6 Relation Between Strict, Rigorous and Conservative 2PL

Deadlock

Definition

'Deadlock is said to occur when each transaction T in a set of two or more transactions is waiting for some item that is locked by some another transaction T' in the set'.

4.8 MULTIPLE GRANULARITY

- It is better if we can group several data items and consider them as an individual units instead of considering each individual data item on which we will perform synchronisation.

Example: Let's make an assumption that a transaction T_1 demands access over the whole database utilising the concept of locking protocol. Each item in the database must be locked by T_i. This process is time consuming. In place of this, we can allow T_i to issue a single lock acquisition plea. Differently, if transaction T_j wants to access only few data items, then no need to lock the entire database.

- Hence, there exists scope to establish the idea of multiple granularity.
- We can achieve this by allowing various size data items and by defining a hierarchy of data granularities.
- We can represent this hierarchy as a tree.
- A non-leaf node represents the data associated with its descendants.

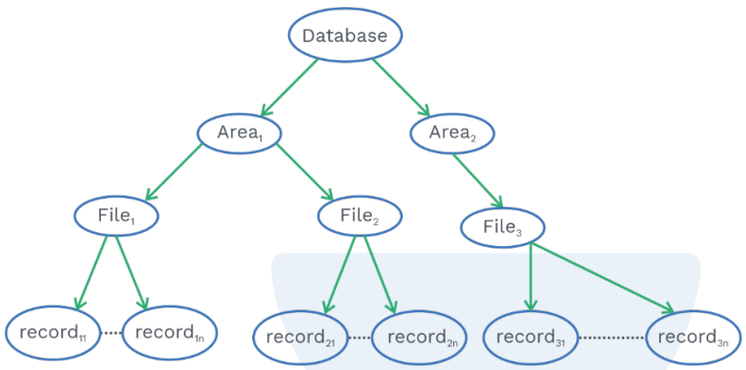


Fig. 4.7 Granularity Hierarchy

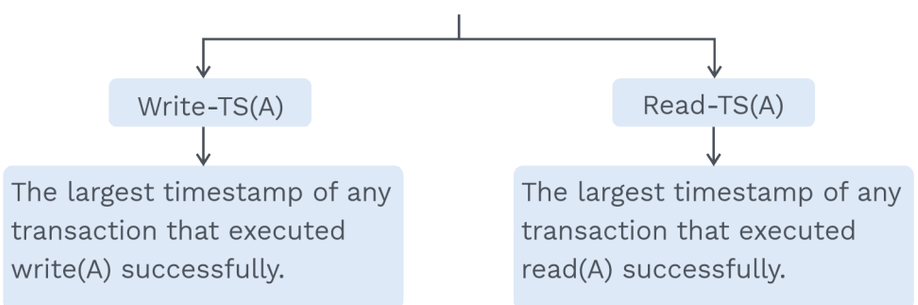
4.9 TIMESTAMP BASED PROTOCOLS

Note:

If we provide a timestamp to each transaction, then we can prevent deadlock.

Definition: 'Transaction's Timestamp is a unique identifier assigned to each transaction'.

- We assign timestamp to the transaction based on when a transaction has arrived.
  TS (T_1) < TS (T_2) means transaction T2 comes after T1.
- The serializabiltiy order can be found using the transaction timestamp.
- We associate 2 timestamp values with each data item A.



- Timestamps will get updated if new write or read instruction is executed.

Timestamp ordering protocol

- We know that timestamp ordering based concurrency control techniques do not use locks.
  Therefore no deadlock will occur.
- In this protocol any conflicting write or read operation will execute in timestamp order.
- If it is not then such an operation is rejected and the transaction will be rolled back.

Note:

Whenever the concurrency control scheme rolls back a transaction, the system provides it with a new timestamp and restarts it.

Example: Consider a schedule:

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

Suppose TS (T_1) = 1 and TS (T_2) = 2 then conflicting action $R_1(A) \rightarrow W_2(A)$ is allowed as TS (T_1) < TS (T_2) but conflicting action $W_2(A) \rightarrow W_1(A)$ is not allowed as transaction having greater timestamp has already executed. So, it will be rolled back and restarted again with different timestamp.

Example: Consider an example:

Suppose TS (T_1) = 1 and TS (T_2) = 2 and following schedule is given.

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

In this example there is no conflict operation from transaction T_1 to T_2 but there is a conflict operation from transaction T_2 to T_1. Since TS (T_1) < TS (T_2) so, conflict operation from T_2 to T_1, i.e. $R_2(A) \rightarrow W_1(A)$ is not allowed. Thus, this transaction T_1 will rollback and restart again with a new timestamp. Let us say TS (T_2) = 2 and TS (T_1) = 3.

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

T_2T_1 is equivalent serial schedule.

Note:

In basic timestamp order protocol, if there are two conflict operations that occur in the incorrect order, then we can reject later of the two atomic operation through abort of the targeted transaction.
So, conflict serializability is secured through timestamp ordering.

Strict timestamp ordering

- A strict timestamp ordering ensures strict and serializable schedule.

Definition

- 'In strict timestamp ordering, a transaction T that issues a read_item (A) or write_item (A) such that TS (T) > write_TS (A) has read or write operation delayed until the transaction T' that wrote the value of A has committed or aborted'.
- The strict timestamp ordering avoids deadlock.

4.10 DEADLOCK HANDLING

There are different ways to handle a deadlock, for example, deadlock prevention, deadlock detection and recovery.

Deadlock prevention

Note:

There prevails two mechanisms to ascertain deadlock prevention: wait-die, wound-wait.

Definition

Time-stamp: It is represented as TS(T_i) and used to prevent deadlock. Transaction timestamp is a unique identifier that is assigned to each transaction.

1) Wait-die

- If TS (T_m) < TS (T_n), means T_m is older than T_n.

Case 1: When an older (T_m) transaction tries to lock an element that is already locked by younger (T_n) transaction then the older (T_m) transaction has to wait.
Otherwise,
Case 2: When younger (T_m) transaction tries to lock an element that is already locked by older (T_n) transaction then the younger (T_m) transaction dies.

2) Wound wait

- If TS (T_i) < TS (T_j), means T_i is older than T_j.

Case 1: When an older transaction (T_i) tries to lock an element that is already locked by younger transaction (T_j) then T_i wounds T_j.
Otherwise,
Case 2: When a younger transaction (T_i) tries to lock an element which is already locked by older transaction (T_j) then T_i has to wait.

- In both methods of preventing deadlock, younger of the two transactions where the deadlock is present, get aborted.
- Both techniques are deadlock free as no cycle is possible in any of these techniques.

Deadlock detection and recovery (wait for graph)

Definition

'In deadlock detection, system checks whether the deadlock exists or not'.

- It helps in the detection of deadlock existence.
- A directed edge will be formed if transaction T_1 is waiting to lock an item that is currently locked by transaction T_2.
- Remove the directed edges from the graph if the lock is released by T_j on those items for which T_i was waiting.
- Deadlock is present in the wait for graph if and only if graph has a cycle.

Example: Consider the figure given below:

$T_1 \xrightarrow{\text{Waiting for}} T_2, T_3$

$T_3 \xrightarrow{\text{Waiting for}} T_2$

$T_2 \xrightarrow{\text{Waiting for}} T_4$

$T_4 \xrightarrow{\text{Waiting for}} T_3$

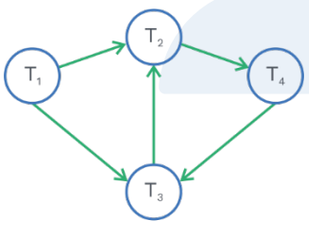


Diagram representing wait for graph with a cycle.

Graph-based protocols

- There is a graph-based protocol that does not use 2PL.

It is also called as tree protocol.

Note:

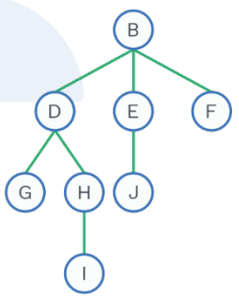
The main advantage of the tree protocol is that we can completely avoid deadlocks.

- We only use exclusive locks in the tree protocol.
- Any data item can be locked at most once by each transaction T_i.
- These are the following rules:

1) Any transaction, T_i is permitted to submit lock acquisition request on any data item.
2) T_i is permitted to lock data item, X if it has lock grant over parent of X.
3) Lock release can occur at any time instant.
4) The transaction T_i cannot be granted lock on the same data item after unlocking it previously.

Example: Consider the given below schedule:

T_1	T_2	T_3	T_4
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock-X(D)		
lock-X(E)			
lock-X(D)			
unlock (B)			
unlock (E)			
		lock-X(B)	
		lock-X(E)	
	unlock (H)		
lock-X(G)			
unlock (D)			
			lock-X(D)
			lock-X(H)
			unlock (D)
			unlock (H)
		unlock (E)	
		unlock (B)	
unlock (G)			



Thomas write rule

Thomas write rule rejects fewer write operations by modifying the checks for the write_item (Q) operation as follows:

1)	If read_TS (Q) > TS (T), then abort and roll back T and reject the operation.
2)	If write_TS (Q) > TS (T), then do not execute the write operation but continue processing.
3)	If none of the above two condition occurs, then execute the write_ item (Q) operation of T and set write_TS (Q) to TS (T).

Example: Schedule are given below:

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

Here, in Thomas write rule, $W_2(A) \rightarrow W_1(A)$ is allowed, no rollback.

Note:

T_1 and T_2 are transactions such that time stamp of T_1 < time stamp of T_2. Then:

	Not Allowed	Allowed
Basic time stamp ordering protocol (similar to conflict serializable)	$R_2(A) \rightarrow W_1(A)$ $W_2(A) \rightarrow R_1(A)$ $W_2(A) \rightarrow W_1(A)$	$R_2(A) \rightarrow R_1(A)$
Thomas write time stamp ordering protocol (similar to view serializable)	$R_2(A) \rightarrow W_1(A)$ $W_2(A) \rightarrow R_1(A)$	$R_2(A) \rightarrow R_1(A)$ $W_2(A) \rightarrow W_1(A)$

Chapter Summary

- **Transaction:** A collection of operations that forms a single logical unit of work.
- **ACID properties.**

  There are 4 ACID properties that a transaction needs to follow:

1)	A	⟶	Atomicity
2)	C	⟶	Consistency
3)	I	⟶	Isolation
4)	D	⟶	Durability

- **Types of failure in a system:** Transaction failure, system crash, disk failure, power failure, software crash, natural hazards, etc.
- **Transaction states:** There are five states for a transaction namely active, partially committed, failed, committed, aborted through which a transaction goes in its lifetime.
- Transaction processing system allows multiple transaction to run concurrently.
- **Concurrency problems in transactions:**
  1) Reading uncommitted data (W-R)
  2) Unrepeatable read (R-W)
  3) Overwriting uncommitted data (W-W)
  4) Phantom read problem
- **Serial schedule:** Serial schedule are those schedule where operations of each transaction executes consecutively.
- **Complete schedule:** When the last operation of each transactions is commit or abort the operation. That schedule is known as complete schedule.
- **Serializability:** Serializability is a property that indicates the correctness of the schedule when a concurrent transactions are executing.
- **Types of schedule based on serializability:**
  1) Conflict serializable schedule.
  2) View serializable schedule.
- **Types of schedule based on recoverability:**
  1) Irrecoverable schedule
  2) Recoverable schedule

3) Cascade rollback recoverable schedule
4) Strict recoverable schedule

- Recoverable schedules ensures that, once a transaction commits, it never rollbacks.
- If one transaction failure causes multiple transactions to rollback, it is called as cascading rollback.
- A serializability order of the transactions can be obtained through topological sorting of precedence graph.
- There are different type of concurrency control techniques such as lock-based protocols, timestamp based protocol.
- **Deadlock:** When none of the transactions is able to proceed with its normal execution, this situation is known as deadlock.
- **Two techniques to handle deadlock:**
  1) Deadlock prevention (wait-die and wound wait)
  2) Deadlock detection and recovery
- To prevent deadlock, there is a concept called as transaction timestamp denoted as TS (T_i) which is a unique identifier assigned to each transaction.