Transaction processing, Concurrency control and Recovery Management: transaction model properties, state serializability, lock base protocols, two phase locking.

Transactions and Concurrency

Transactions

A transaction is a set of operations (for selecting and updating various data items) treated as a unit so that either all of them complete successfully, or none of them execute (***all or nothing***). During transaction execution the database may be inconsistent but after completing database will be back to consistent state.

When more than one user concurrently access and update same data source, transaction helps to maintain the consistency of data source.

A DBMS must ensure **four important properties** of transactions to maintain data consistency during concurrent execution of transaction and system failure. In short this is called **ACID** property. The ACID properties allow safe sharing of data.

a. **Atomicity**: Either all operations of the transaction are carried out or none are.
b. **Consistency**: Each transaction must preserve the consistency of DBMS. Any work in progress must not be visible to other concurrent transactions until the transaction has been committed.
c. **Isolation**: A transaction should appear to be running by itself, the effects of other concurrent transactions must be invisible to this transaction, and the effects of this transaction must be invisible to other concurrent transaction.
d. **Durability**: When the transaction is committed, it must be persisted so it is not lost in the event of any system failure. Only committed transaction are recovered during power-up and crash recovery; uncommitted work is roll back.

DBMS ensures the above properties by logging all actions in following logs :

- **Undo** - the actions of incomplete transactions
- **Redo** - actions of committed transactions not yet propagated to disk

Example of Fund Transfer

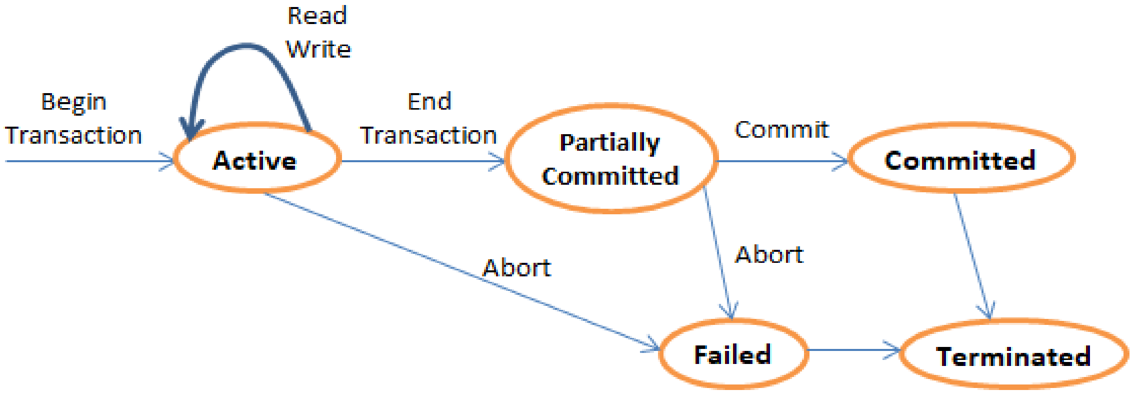
Transaction to transfer $50 from account A to account B:

1. **read(A)** Select Balance of account A (say BalanceA)
2. BalanceA := BalanceA – 50
3. **Update** the balance of A with BalanceA - 50
4. **Select** Balance of account B (say BalanceB)
5. BalanceB := BalanceB + 50
6. **Update** the balance of B with BalanceB + 50

- Consistency requirement – the sum of A and B is unchanged by the execution of the transaction.
- Atomicity requirement — if the transaction fails after step 2 and before step 5, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
- Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.
- Isolation requirement — if between steps 2 and 5, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum A + B will be less than it should be).

Transaction State and additional Operations

Following figure shows a state transition diagram that describes how a transaction moves through its execution states. For recovery purposes, the DBMS needs to keep track of when the transaction starts, terminates, and commits or aborts. The recovery manager keeps track of the following operations:



- BEGIN_TRANSACTION: This marks the beginning of transaction execution. After this operation transaction is in **active** state (the initial state); the transaction stays in this state while it is executing.
- READ OR WRITE: These specify read or write operations on the database items that are executed as part of a transaction when it is in active state.

- END_TRANSACTION: This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted for some reason. At this stage the transaction is in **Partially Committed** state, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may prohibit its successful completion.

- COMMIT_TRANSACTION: This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone. The transaction is in **committed** state. At this point, some recovery mechanisms in place to ensure the durability of committed transaction in case of system failure.

- **ROLLBACK (OR ABORT**): This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone. The transaction will be in **failed** state. A transaction can go to the failed state if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.

- The **terminated** state corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transactions may be restarted later-either automatically or after being resubmitted by the user-as brand new transactions.

Concurrent Execution of Transactions

In a multiprogramming environment multiple transactions are allowed to run due to following advantages:

- Increased processor and disk utilization, leading to better transaction throughput: one transaction can be using the CPU while another is reading from or writing to the disk reduced average response time for transactions.

- Interleaved execution of a short transaction (need less time to complete) with a long transaction usually allows the short transaction to complete quickly. In serial execution, the short transaction may get stuck behind a long transaction, leading to unpredictable delays in **response time**.

Concurrent transactions only interact with each other via database read and write operations, they are not allowed to exchange message. When concurrent transactions share same data source, it can cause the database to become inconsistent, even when the transactions individually preserve correctness of the state, and there is no system failure.

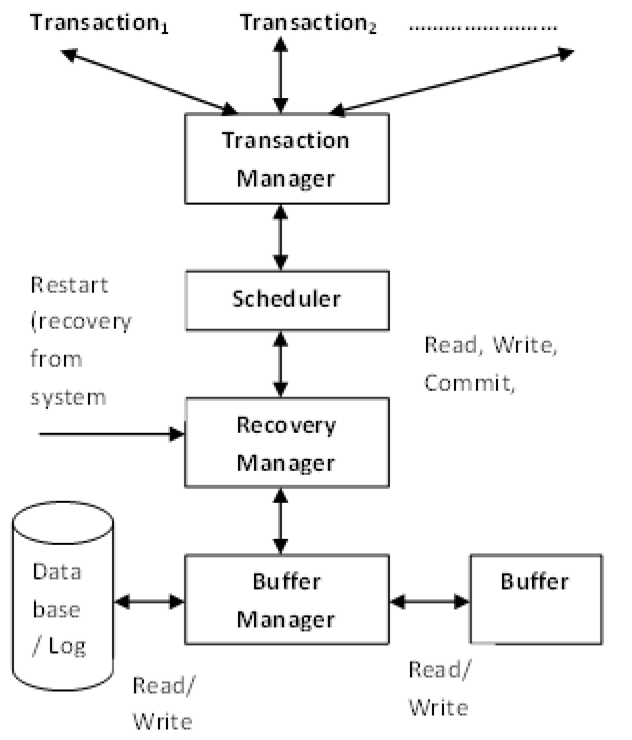
Thus the order in which the individual action of different transactions occur needs to be regulated in some manner.

- The general process of assuring that transactions preserve consistency when executing simultaneously is called **concurrency control**.

- The **scheduler component of the DBMS** is performing this activity.

Database System Architecture

Modules of DBMS which are responsible for concurrency control and recovery are shown here.



- **Recovery Manager** is responsible for ensuring that the effects of all committed transactions are reflected on the database. Similarly any effect of aborted transaction will be undone.

- **Scheduler** module is responsible for controlling the order in which transaction operations would be executed. In order to control the execution sequence of transaction operations, the scheduler may delay a particular operation by placing it in a queue or may reject it (aborting transaction).

- **Buffer Manager –** to control the data transfer between database (residing in the disk) and the main memory. In order to avoid a disk access for every read and write operation, usually data transfer takes place blocks.

Thus the execution of Read(X,x) by a transaction involves following steps :

X – Data Item
x – Local variable

a. Get the address of the disk block that contains the data item X.
b. If the block contains X is not already present in the buffer, bring the disk block into the buffer.
c. Copy the data item X from the buffer into the local variable ‘x’ of the transaction.

Similarly, Write(X,x) is executed as follows:

a. Find the address of the disk block that contains the data item X.
b. If the block contains X is not already present in the buffer, bring it into the buffer.
c. Write the value of the local variable ‘x’ into the buffer location corresponding to X
d. The updated buffer block is stored back into the disk, either immediately, or at a later time.

Note that the actual update of the database takes place when the updated buffer is stored in the disk in step d. Since a buffer block may contain more than one data item, after a single update often it is not immediately stored back. A buffer block which contains one or more updated data item is called a **dirty block** (or page). When a dirty block is to be stored back, (also called flushed) depends on the recovery manager and the cache replacement policy used. Some well know replacement strategies are, **least-recently-used (LRU), FIFO** etc.

Concurrency control

Concurrency control is the management of contention (conflict) for data resources. There are mainly two **concurrency control schemes** which help to achieve the consistency of DB.

Take an Example to understand two schemes.

Say two users A and B want to update customer #123. As B is updating after A we expect B's update must persist. If sequence is as shown below then we encounter following problem.

1. User A reads the row for customer #123
2. User B reads the row for customer #123
3. User B updates the row for customer #123
4. User A updates the row for customer #123 *and overwrites User B's changes*

The problem here is that we'll lose User B's changes without even noticing. So, DB will be in inconsistence state after committing both transactions.

So we need to control the concurrency to avoid such inconsistency.

1. ***Pessimistic scheme*** : it locks a given resource early in the data access transaction and does not release it until the transaction is closed.

   Here the database server lock the row on User A's behalf, so User B has to wait until User A has finished its work before proceeded. We effectively solve the problem by **not allowing two users to work on the same piece of data at the same time**. It just prevents the conflict.

   A user is thinking "I'm sure someone will try to update this row at the same time as I will, so I better ask the database server to not let anyone touch it until I'm done with it."

   It is more efficient when update collisions are expected to occur often.

2. ***Optimistic scheme*** : when locks are acquired and released over a short period of time at the end of a transaction.

   The optimistic scheme doesn't actually lock anything (at the beginning) – instead, it asks User A to remember what the row looked like when he first saw it, and when it's time to update it, the user asks the database to go ahead only if the row still looks like he remembers it. It doesn't prevent a possible conflict, but it can detect it before any damage is done and fail safely.

   To enable optimistic concurrency, the columns marked for update and their original values are added explicitly through a **WHERE** clause in the **UPDATE** statement so that the statement fails if the underlying column values have been changed. As a result, this scheme can provide **column-level concurrency control**; pessimistic schemes can control concurrency at the **row level** only.

Optimistic schemes typically perform this type of test only at the end of a transaction. If the underlying columns have not been updated since the beginning of the transaction, field's updates are committed and the locks are released. If locks cannot be acquired or if some other transaction has updated the columns since the beginning of the current transaction, the transaction is rolled back: All work performed within the transaction is lost.

The optimistic approach is a user thinking "I'm sure it's all good and there will be no conflicts, so I'm just going to remember this row to double-check later, but I'll let the database server do whatever it wants with the row." It is more efficient when update collisions are expected to be **infrequent**.

The pros and cons

Both pessimistic and optimistic concurrency control mechanisms provide different performance, e.g., the different average transaction completion rates or throughput.

Strength	Weekness
Pessimistic concurrency	
• Guarantee that all transactions can be executed correctly and Database is always in consistent state.	• Transactions are slow due to the delay by locking or time-stamping event. • Transaction latency (inactivity) increases significantly. • Throughput or the amount of work (e.g. read/write, update, rollback operations, etc.) is reduced.
Optimistic concurrency	
• Transactions are executed more efficiently. • Data content is relatively safe. • Throughput is much higher.	• There is a risk of data interference among concurrent transactions since it transactions conflict may occur during execution. In this case, data is no longer correct. • Database may have some hidden errors with inconsistent data; even conflict check is performed at the end of transactions. • Transactions may be in deadlock that causes the system to hang.

Scheduling Transactions for concurrent execution

Schedule (**Histories**) – A chronological execution sequence of a transaction is called a schedule. A schedule can have many transactions in it, each comprising of a number of actions. So a schedule is a list of actions (reading, writing, aborting, or committing) from a set of concurrent transactions subject to following constraints. Say we are defining a schedule (or history) S of n transactions T_1 , T_2, ... , T_n which follow the constraints stated below :

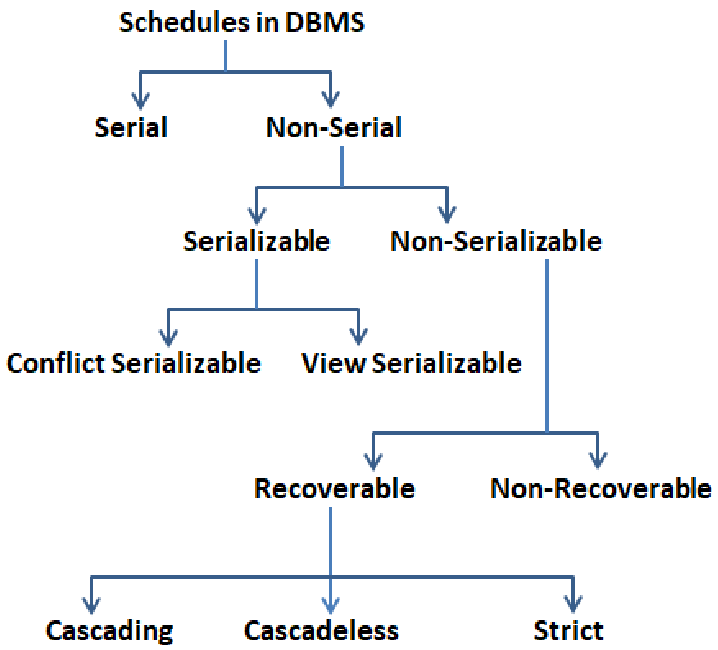
- For each transaction T_i in S, the operations of T_i in S must appear in the same order in which they occur in T_i
- The operations from other transactions T_j can be interleaved with the operations of T_i, in S.

In a multi-transaction environment, serial schedules are considered as a benchmark.

The execution sequence of an instruction in a transaction cannot be changed, but two transactions can have their instructions executed in a random fashion. The DBMS interleaves the actions of different transactions to improve performance.

This execution does no harm if two transactions are mutually independent and working on different segments of data; but in case these two transactions are working on the same data, then the results may vary. This ever-varying result may bring the database to an inconsistent state. **So not all interleaving should be allowed**.

Different Types of Schedules



We will discuss what interleaving, or schedules, a DBMS should allow.

Some commonly used notations to denote the actions performed by transaction :

R_T (O) – Reading a data object (O) by the transaction (T)
W_T (O) – Writing a data object (O) by the transaction (T)
Abort $_T$ **–** Aborting the transaction (T), **Commit** $_T$ **–** Committing the transaction (T)

Complete Schedule : A schedule S of n transactions TI , T2, ••• , Tn' is said to be a complete schedule if the following conditions hold:

1. The operations in S are exactly those operations in TI , T2, •.• , Tn' including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction T_i, their order of appearance in S is the same as their order of appearance in T_i;
3. For any two conflicting operations, one of the two must occur before the other in the schedule.

Anomalies with Interleaved Execution

If interleaving of operations is allowed, there will be many possible orders in which the system can execute the individual operations of the transactions. Some of these orders may leave a database in inconsistent state.

There are three main ways in which a schedule involving two transactions may leave a database in inconsistent state. Three anomalies are described below for two transactions T1 and T2 conflict with each other.

1. **Reading Uncommitted Data (WR Conflicts) :**

C: Commit, A : Abort

→ **time** **T1 :** R(A), W(A) R(B),W(B), A **T2 :** R(A), W(A), C	T2 is reading a database object (A) modified by T1, before T1 commits. Such a read is called **dirty read**.

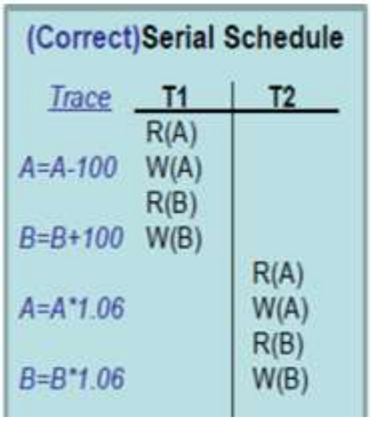
Example :

To illustrate the WR-conflict consider the following problem:

T1: Transfer $100 from Account A to Account B
T2: Add the annual interest of 6% to both A and B.

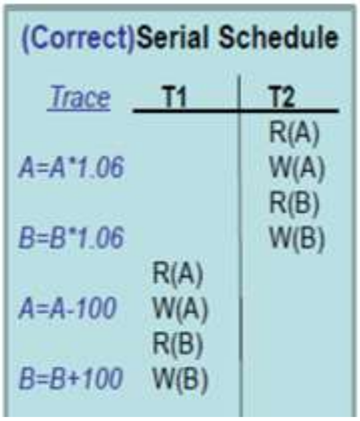
Problem : Account B was credited with the interest on a smaller amount (i.e., 100$ less), thus the result is not equivalent to the serial schedule.

Either you first transfer the amount and then apply interest or first apply the interest and then transfer to maintain consistency. In the following serial schedule we are doing that.



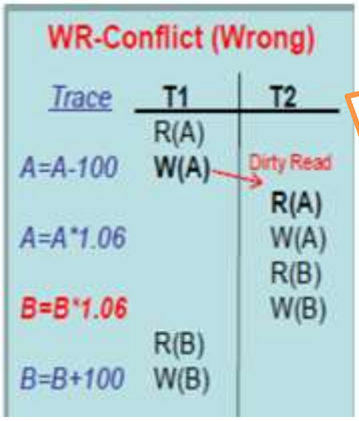
(Correct)Serial Schedule

Trace	T1	T2
	R(A)	
A=A-100	W(A)	
	R(B)	
B=B+100	W(B)	
		R(A)
A=A*1.06		W(A)
		R(B)
B=B*1.06		W(B)



(Correct)Serial Schedule

Trace	T1	T2
		R(A)
A=A*1.06		W(A)
		R(B)
B=B*1.06		W(B)
	R(A)	
A=A-100	W(A)	
	R(B)	
B=B+100	W(B)	



WR-Conflict (Wrong)

Trace	T1	T2
	R(A)	
A=A-100	W(A)	Dirty Read
		R(A)
A=A*1.06		W(A)
		R(B)
B=B*1.06		W(B)
	R(B)	
B=B+100	W(B)	

For A Interest after transfer
For B Interest before transfer.

2. **Unrepeatable (fuzzy) Reads (RW Conflicts) :**

→ **time** **T1 :** R(A), R(A),W(A), C **T2 :** R(A), W(A), C A transaction rereads data it has previously read and finds that another committed transaction has modified or deleted it.	T2 changes the value of a database object (A) that has been read by T1, while T1 is still in progress. During second read T1 will get different value, even though it has not modify A in the meantime. It discovers two different versions of A, and T1 would be forced to abort (T1 would not know what to do)
Example : Suppose a student goes to the library and requests a book A from a library assistant. If the book is available (no. > 0) it will be issued. The library assistant checks the availability and informs that book is available. But before issuing the book to that student, another student submits another request for the same book A. Another library assistant checks the availability and issues the book. If 1st assistant try to issue the book, it will be a false issue.	

3. Overwriting Uncommitted Data (WW Conflicts) :

<table>
<tr>
<td>→ <b>time</b>
<b>T1 :</b> W(A), R(A), C
<b>T2 :</b> W(A), W(B), C

The problem is that we have lost update.</td>
<td>T2 overwrite the value of a database object (A) which has already been modified by T1, while T1 is still in progress.

During second read T1 will get different value, even though it has not modified A in the meantime.</td>
</tr>
<tr>
<td colspan="2">Example : Suppose two employee A and B must get same salary (required consistency)
Say T1 sets their salary to Rs 5000 and T2 sets their salary to Rs 8000. If the serial order is T1, T2 the end result is both will get Rs 8000. If the serial order is T2, T1 the end result is both will get Rs. 5000. In both cases consistency (both will get same salary) is maintained. But consider following Interleaving of actions of T1 and T2ed

T1 : W(A) W(B) → Set A=5000, B=5000
T2 : W(B) W(A) → Set B=8000, A=8000

So the consistency is violated at the end of transactions (salary is not equal)</td>
</tr>
</table>

Phantom reads: A transaction re-executes a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

Conflict Operation : Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions:

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

These three conditions are applicable for all the above anomalies described above.

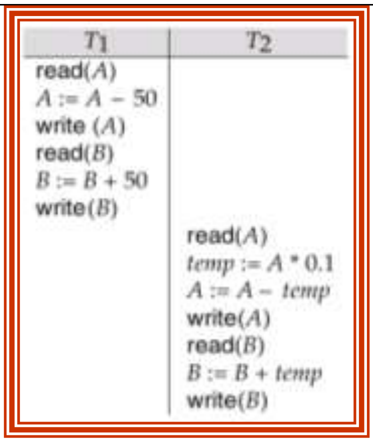
Example :

1. **WR Conflict** (dirty read): A transaction T2 could read a database object A that has been modified by another transaction T1, which has not yet committed.
2. **RW Conflict** (unrepeatable read): A transaction T2 could change the value of an object A that has been read by a transaction T1, while T1is still in progress.
3. **WW Conflict** (lost update): A transaction T2 could overwrite the value of an object A, which has already been modified by a transaction T1, while T1 is still in progress.

Example Schedules to explain above anomalies

Let T1 transfer $50 from A to B, and T2 transfer 10% of the balance from A to B. The following is a **serial schedule**, in which T1 is followed by T2.

Start with A=100 B=200

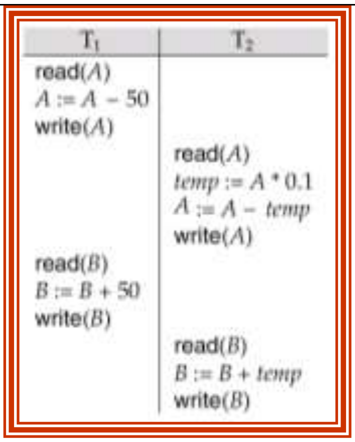


	T_1	T_2	
A= 100	read(A)		
	A := A − 50		
A= 50	write (A)		
B= 200	read(B)		
	B := B + 50		
B= 250	write(B)		
		read(A)	A=50
		temp := A * 0.1	
		A := A − temp	
		write(A)	A=45
		read(B)	B= 250
		B := B + temp	
		write(B)	B = 255

We end with A=45 B=255

Let T1 and T2 be the transactions defined as per serial schedule. The following schedule is not a serial schedule, but it is equivalent to the previous serial schedule

Start with A=100 B=200

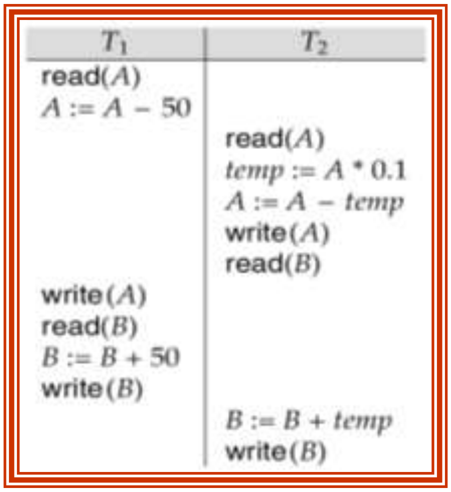


	T_1	T_2	
A= 100	read(A)		
	A := A − 50		
A= 50	write(A)		
		read(A)	A=50
		temp := A * 0.1	
		A := A − temp	
		write(A)	A=45
B=200	read(B)		
	B := B + 50		
B= 250	write(B)		
		read(B)	B=250
		B := B + temp	
		write(B)	B=255

We end again with A=45 B=255 In both Schedules, the sum A + B is preserved

The following concurrent schedule does not preserve the value of the sum A + B – The schedule is wrong and should not be allowed.

Lets start with A=100 B=200



	T_1	T_2	
A= 100	read(A)		
	A := A − 50		
		read(A)	A= 100
		temp := A * 0.1	Temp = 10
		A := A − temp	
		write(A)	A= 90
		read(B)	B = 200
A= 50	write(A)		
B= 200	read(B)		
	B := B + 50		
B= 250	write(B)		
		B := B + temp	
		write(B)	B = 210

We end with A=50 B=210

Before execution of the schedule
A+B = 100+200
After execution
A+B = 50+210

Serializibility and Serializable Schedule

The database system must control concurrent execution of transactions, to ensure consistency. Before we examine how the database system can carry out this task, we must first understand which schedules will ensure consistency, and which schedules will not.

To understand any concurrent control scheme the concept of serializability should be clear.

Basic Assumption – Each transaction preserves database consistency. Thus serial execution of a set of transactions preserves database consistency.

A schedule is serializable if it is ***equivalent*** to a serial schedule. Concurrent execution of transactions in serializable schedule should provide the same result and have the same effect on the DB as it is produced by equivalent serial execution of the same transactions. A schedule that ensures this property is called a **serializable schedule**.

The concept of serializability of schedules is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules.

→ time

T1 :	R(A), W(A)		R(B),W(B)		C
T2 :		R(A), W(A)		R(B), W(B),C	

Note: If each transaction preserves consistency, every serializable schedule preserves consistency

As an example, the schedule shown here is serializable. Even though the actions of T1 and T2 are interleaved, the result of this schedule is equivalent to running T1 (in its entirety) and then running T2. Intuitively, T1 's read and write of B is not influenced by T2's actions on A, and the net effect is the same if these actions are 'swapped' to obtain the serial schedule Tl; T2.

The preceding definition of a serializable schedule does not cover the case of schedules containing aborted transactions.

Example : Consider a banking system, where two transactions T1 and T2 are involved in fund transfer from an account A1 to B1.

Serial execution of T1 and T2 in any order.		Concurrent execution schedule of T1 and T2.	
Say Initial balance A1 = **1500**, B1 = **1000**		**T1**	**T2**
		Read(A1, x1)	
T1	**T2**	x1:= x1 -200	
Read(A1, x)	Read(A1, x)		Read(A1, x2)
x:= x -200	x:= x -800		x2:= x2 -800
Write(A1, x)	Write(A1, x)		Write(A1, x2)
Read(B1, y)	Read(B1, y)		Read(B1, y2)
y:= y+200	y:= y+800		

Write (B1, y) Write (B1, y) If we run T1 then T2, after T1 A=1300, B=1200, after T2 A=500, B=2000 Final balance A1 = 500, B1 = 2000 Initial sum of balance = 1500 + 1000 Final sum of balance = 500 + 2000 Thus the sum of A1 and B1 is preserved	Write(A1, x1) Read(B1, y1) y:= y1+200 y2:= y2+800 Write (B1, y1) Write (B1, y2) Final balance A1 = 1300, B1 = 1800 So balance (3100) of A1 and B1 is not preserved.

- For two schedules to be equivalent, the operations applied to each data item affected by the schedules should be applied to that item in both schedules in the same order.

  In above two schedules operation on each data item A1 and B1 are not in same order. Yellow color highlights the T1 and green highlights T2.

  Serial Scheduled : R(A1) → W(A1) →R(B1) → W(B1) → R(A1) →W(A1) → R(B1) → W(B1)
  Concurrent schedule : R(A1) → R(A1) → W(A1) →R(B1) → W(A1) →R(B1) → W(B1) → W(B1)

- Two definitions of equivalence of schedules are generally used: **conflict equivalence** and **view equivalence**. Conflict equivalence is the more commonly used definition. We are not discussing view equivalence here.

Conflict Equivalent

In a schedule two operations are conflicting if they satisfy all

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

Say we have a schedule S running two transactions concurrently, and we can reorder the operations in them and create 2 more schedules S1 and S2.

Two schedule S1 and S2 are **conflict equivalent** when ordering of the conflicting operation in S1 is same as in S2. (Note : Ordering should be maintained only for the conflicting operations). For example, S1 contains T1's R(X) before T2's W(X) , then it should be the same in S2 also.

So the schedules S1 and S2 to be conflict-equivalent if the following conditions are satisfied:

1. Both schedules S1 and S2 involve the same set of transactions (including *ordering of actions* within each transaction).
2. The order of each pair of conflicting actions in S1 and S2 are the same.

If a schedule S1 can be transformed into a schedule S2 by a series of swaps of non-conflicting operations and S1 and S2 produces same consistent database, we say that S1 and S2 are **conflict equivalent.**

Conflict Serializability

- A schedule is said to be conflict-serializable when the schedule is ***conflict-equivalent to one or more serial schedules***. (This means ordering of the conflicting operation in the schedule is same as in one or more serial schedule).

- Another definition for conflict-serializability is that a schedule is conflict-serializable if and only if its **precedence** graph/ **serializability** graph, is acyclic when only **committed** transactions are considered.
  If uncommitted transactions are also included in the graph, then cycles involving uncommitted transactions may occur without conflict serializability violation.

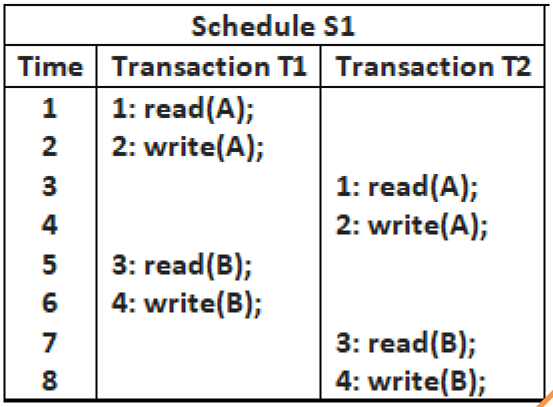
A schedule S1 which interleave instructions of T1 and T2 can be executed if it is equivalent to some serial schedule without losing consistency.

So finding conflict serializibility of an interleaved schedule is a crucial measure to ensure execution of it without causing any inconsistency.

Process of finding conflict Serializability

- We have to swap **non-conflicting** instructions of S1 until we reach a serial schedule. This has to be done with immediate next or previous instruction in a schedule.
- Can we swap the instruction 2 of T1 with 1 of T2? NO. Why? Both instructions are conflict with each other. They both are trying to access the same data item A.
- Can we swap the instruction 3 of T1 with 2 of T2? YES. Why? The instructions Read-Write may be conflict but they are trying to access different data items. Hence, not conflict and permitted.
- We can do series of swaps as follows to arrive at a serial schedule S2;

Let's start with the example schedule S1 as shown below to find if it is **conflict Serializability** :



Schedule S1		
Time	**Transaction T1**	**Transaction T2**
1	1: read(A);	
2	2: write(A);	
3		1: read(A);
4		2: write(A);
5	3: read(B);	
6	4: write(B);	
7		3: read(B);
8		4: write(B);

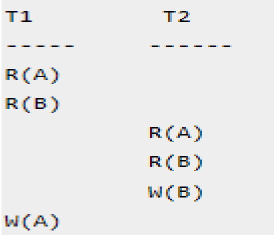
In 1st step we swap Read(B) of T1 with Write(A) of T2 and the result schedule is shown in following table. This table indicates the result schedule after each swap.

Result schedule	Action Taken	Permitted?	Serial?
Time **Transaction** **Transaction**	Read(B) of T1 swapped with	YES Reason: Non-	NO Reason:

<table>
<tr>
<td>
<table>
<tr><th></th><th>T1</th><th>T2</th></tr>
<tr><td>1</td><td>1: read(A);</td><td></td></tr>
<tr><td>2</td><td>2: write(A);</td><td></td></tr>
<tr><td>3</td><td></td><td>1: read(A);</td></tr>
<tr><td>4</td><td>3: read(B);</td><td></td></tr>
<tr><td>5</td><td></td><td>2: write(A);</td></tr>
<tr><td>6</td><td>4: write(B);</td><td></td></tr>
<tr><td>7</td><td></td><td>3: read(B);</td></tr>
<tr><td>8</td><td></td><td>4: write(B);</td></tr>
</table>
</td>
<td>Write(A) of T2</td>
<td>conflict</td>
<td>instructions are interleaved</td>
</tr>
<tr>
<td>
<table>
<tr><th>Time</th><th>Transaction T1</th><th>Transaction T2</th></tr>
<tr><td>1</td><td>1: read(A);</td><td></td></tr>
<tr><td>2</td><td>2: write(A);</td><td></td></tr>
<tr><td>3</td><td>3: read(B);</td><td></td></tr>
<tr><td>4</td><td></td><td>1: read(A);</td></tr>
<tr><td>5</td><td></td><td>2: write(A);</td></tr>
<tr><td>6</td><td>4: write(B);</td><td></td></tr>
<tr><td>7</td><td></td><td>3: read(B);</td></tr>
<tr><td>8</td><td></td><td>4: write(B);</td></tr>
</table>
</td>
<td>Read(B) of T1 swapped with Read(A) of T2</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>
<table>
<tr><th>Time</th><th>Transaction T1</th><th>Transaction T2</th></tr>
<tr><td>1</td><td>1: read(A);</td><td></td></tr>
<tr><td>2</td><td>2: write(A);</td><td></td></tr>
<tr><td>3</td><td>3: read(B);</td><td></td></tr>
<tr><td>4</td><td></td><td>1: read(A);</td></tr>
<tr><td>5</td><td>4: write(B);</td><td></td></tr>
<tr><td>6</td><td></td><td>2: write(A);</td></tr>
<tr><td>7</td><td></td><td>3: read(B);</td></tr>
<tr><td>8</td><td></td><td>4: write(B);</td></tr>
</table>
-
</td>
<td>Write(B) of T1 swapped with Write(A) of T2</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>
<table>
<tr><th>Time</th><th>Transaction T1</th><th>Transaction T2</th></tr>
<tr><td>1</td><td>1: read(A);</td><td></td></tr>
<tr><td>2</td><td>2: write(A);</td><td></td></tr>
<tr><td>3</td><td>3: read(B);</td><td></td></tr>
<tr><td>4</td><td>4: write(B);</td><td></td></tr>
<tr><td>5</td><td></td><td>1: read(A);</td></tr>
<tr><td>6</td><td></td><td>2: write(A);</td></tr>
<tr><td>7</td><td></td><td>3: read(B);</td></tr>
<tr><td>8</td><td></td><td>4: write(B);</td></tr>
</table>
</td>
<td>Write(B) of T1 swapped with Read(A) of T2</td>
<td>YES</td>
<td>YES
T1 then T2 is the serial order</td>
</tr>
</table>

Final answer: schedule S1 is conflict serializable to the serial schedule T1-T2 that is, the concurrent schedule S1 is equivalent to executing T1 first then T2.

Example of Conflict Serializability



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

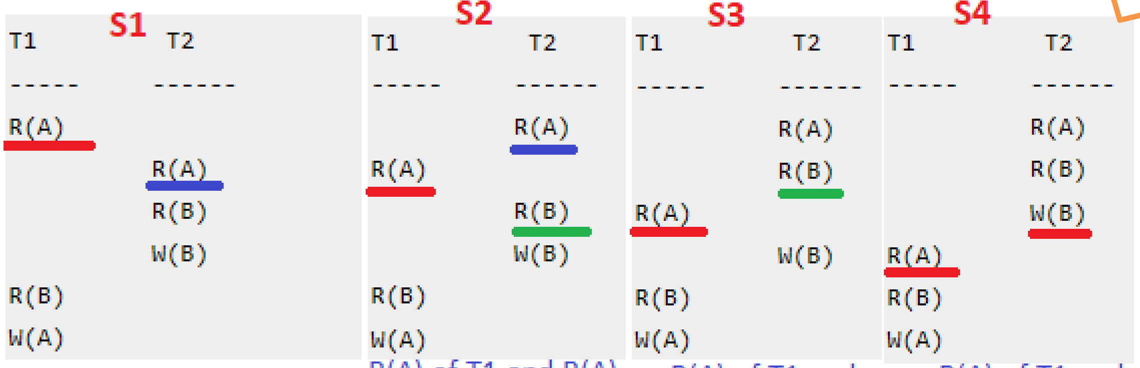
To convert this schedule into a serial schedule we must have to swap the R(A) operation of transaction T2 with the W(A) operation of transaction T1.

However we cannot swap these two operations because they are conflicting operations, thus we can say that this given schedule is **not Conflict Serializable**.

Let’s take another example as shown below. Here we are **swapping non-conflicting operations** of the first schedule (original schedule)

- After swapping R(A) of T1 and R(A) of T2 we get S2
- After swapping R(A) of T1 and R(B) of T2 we get S3
- After swapping R(A) of T1 and W(B) of T2 we get S4

We finally got a serial schedule after swapping all the non-conflicting operations so we can say that the given schedule is **Conflict Serializable**.



S1

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

S2

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

S3

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

S4

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

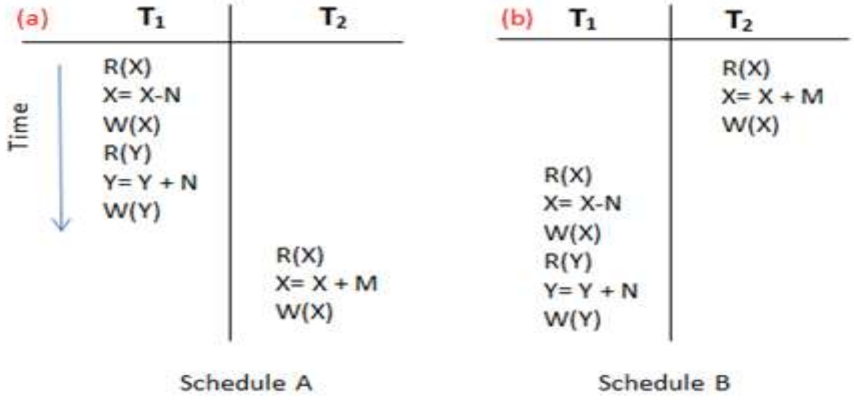
Original Schedule ⇨ R(A) of T1 and R(A) of T2 in S1 is swaped to get S2

R(A) of T1 and R(B) of T2 swaped to get S2

R(A) of T1 and W(B) of T2 swapped to get S4

Following Figure shows example of serial and nonserial schedules involving transactions T1 and T2

(a) Serial Schedule **A** : T1 followed by T2. (b) Serial schedule Serial **B**: T2 followed by T1
(c) Two nonserial schedules **C** and **D** with interleaved operations



(a)

```
T₁              T₂
R(X)
X= X-N
W(X)
R(Y)
Y= Y + N
W(Y)
                R(X)
                X= X + M
                W(X)
```

Schedule A

(b)

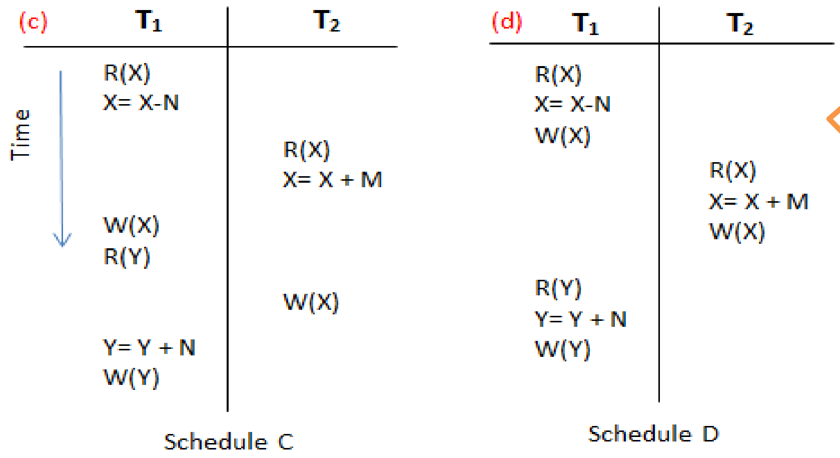
```
T₁              T₂
                R(X)
                X= X + M
                W(X)
R(X)
X= X-N
W(X)
R(Y)
Y= Y + N
W(Y)
```

Schedule B

In Schedule A and D

R(X) of T_2 reads the value of X written by T1 while the other read operations read the database values from the initial database state.

In addition, T1 is the last transaction to write Y, and T2 is the last transaction to write X in both schedules.



So schedule D of above Figure is equivalent to the serial schedule A. Because A is a serial schedule and schedule D is equivalent to A, D is a serializable schedule.

Because A is a serial schedule and schedule D is equivalent to A, D is a serializable schedule.

In Schedule D, we can move $r_1(Y)$, $w_1(Y)$ of T1 before $r_2(X)$, $w_2(X)$ of T2, since they access different data items leading to the equivalent serial schedule T_1, T_2.

Schedule C of above Figure is not equivalent to either of the two possible serial schedules A and B, and hence is not serializable. Trying to reorder the operations of schedule C to find an equivalent serial schedule fails, because r2(X) and w1 (X) conflict, which means that we cannot move r2(X) down to get the equivalent serial schedule T1, T2. Similarly, because W1 (X) and w2(X) conflict, we cannot move w1(X) down to get the equivalent serial schedule T2,T1.

Test for conflict serializability

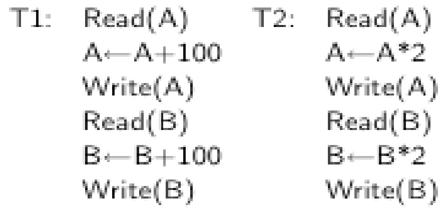
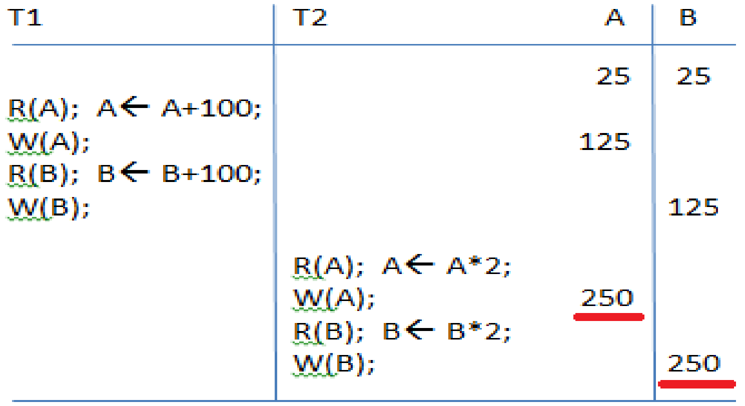
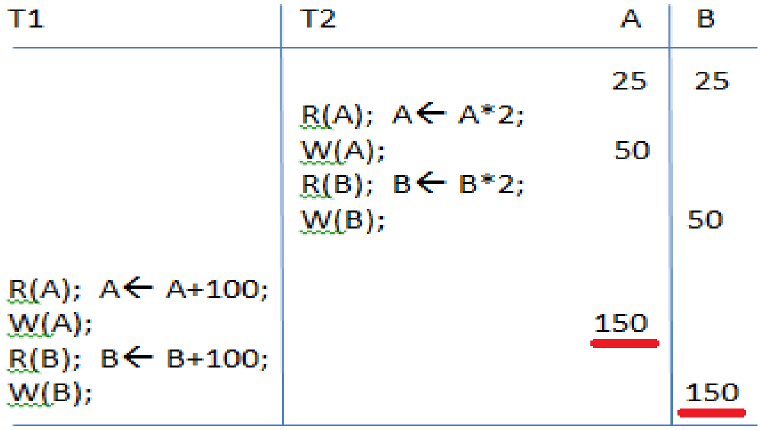
We now present a simple and efficient method (algorithm) for determining conflict serializability of a schedule.

Consider a schedule S. We construct a directed graph, called a precedence graph, fromS. This graph consists of a pair G =(V, E), where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions participating in the schedule. Now we draw the set of edges between two vertices $T_j \rightarrow T_k$ using following steps :

Step	Operation in T_j		Operation in T_k	
1.	For each transaction T_k participating in schedule S, create a node labeled T_k in the precedence graph.			
2.	T_j executes W(X)	Before	T_k executes R(X)	Create edge $T_j \rightarrow T_k$
3.	T_j executes R(X)	Before	T_k executes W(X)	Create edge $T_j \rightarrow T_k$
4.	T_j executes W(X)	Before	T_k executes W(X)	Create edge $T_j \rightarrow T_k$
5.	The schedule S is serializable if and only if the precedence graph has no cycles.			

A schedule is conflict serializable if it produces an acyclic precedence graph. If the precedence graph for S has a cycle, then schedule S is not conflict serializable.

Some examples:

<table>
<tr>
<td>We have the following two transactions that operate on the balance of two bank accounts. The transactions are concurrent, but the assumption is that once the transactions are executed, the balances are the same, that is A=B</td>
<td>
<table>
<tr><td>T1:</td><td>Read(A)
A←A+100
Write(A)
Read(B)
B←B+100
Write(B)</td><td>T2:</td><td>Read(A)
A←A*2
Write(A)
Read(B)
B←B*2
Write(B)</td></tr>
</table>
</td>
</tr>
<tr>
<td colspan="2"><b>Following schedules are obtained by rearranging the order of the actions in T1 and T2.</b></td>
</tr>
<tr>
<td><b>SCHEDULE A: A consistent serial schedule</b>
A serial schedule that is also consistent, is obtained by executing T2 right after T1</td>
<td>
<table>
<tr><th>T1</th><th>T2</th><th>A</th><th>B</th></tr>
<tr><td></td><td></td><td>25</td><td>25</td></tr>
<tr><td>R(A); A← A+100;</td><td></td><td></td><td></td></tr>
<tr><td>W(A);</td><td></td><td>125</td><td></td></tr>
<tr><td>R(B); B← B+100;</td><td></td><td></td><td></td></tr>
<tr><td>W(B);</td><td></td><td></td><td>125</td></tr>
<tr><td></td><td>R(A); A← A*2;</td><td></td><td></td></tr>
<tr><td></td><td>W(A);</td><td>250</td><td></td></tr>
<tr><td></td><td>R(B); B← B*2;</td><td></td><td></td></tr>
<tr><td></td><td>W(B);</td><td></td><td>250</td></tr>
</table>
</td>
</tr>
<tr>
<td><b>SCHEDULE B: Another serial schedule</b>
Another serial schedule is obtained by executing T1 right after T2</td>
<td>
<table>
<tr><th>T1</th><th>T2</th><th>A</th><th>B</th></tr>
<tr><td></td><td></td><td>25</td><td>25</td></tr>
<tr><td></td><td>R(A); A← A*2;</td><td></td><td></td></tr>
<tr><td></td><td>W(A);</td><td>50</td><td></td></tr>
<tr><td></td><td>R(B); B← B*2;</td><td></td><td></td></tr>
<tr><td></td><td>W(B);</td><td></td><td>50</td></tr>
<tr><td>R(A); A← A+100;</td><td></td><td></td><td></td></tr>
<tr><td>W(A);</td><td></td><td>150</td><td></td></tr>
<tr><td>R(B); B← B+100;</td><td></td><td></td><td></td></tr>
<tr><td>W(B);</td><td></td><td></td><td>150</td></tr>
</table>
</td>
</tr>
</table>

SCHEDULE C: A schedule that IS NOT SERIAL but is still consistent

Obtained by interleaving the actions of T1 with those of T2

The order of read and write operations in the schedule is as follows :

r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)

T1	T2	A	B
		25	25
R(A); A← A+100;			
W(A);		125	
	R(A); A← A*2;		
	W(A);	250	
R(B); B← B+100;			
W(B);			125
	R(B); B← B*2;		
	W(B)		250

SCHEDULE D: A schedule that is NOT SERIAL and NOT CONSISTENT

Obtained by interleaving the actions of T1 with those of T2. It is inconsistent because it violates the constraint A=B

T1	T2	A	B
		25	25
R(A); A← A+100;			
W(A);		125	
	R(A); A← A*2;		
	W(A);	250	
	R(B); B← B*2;		
	W(B);		50
R(B); B← B+100;			
W(B);			150

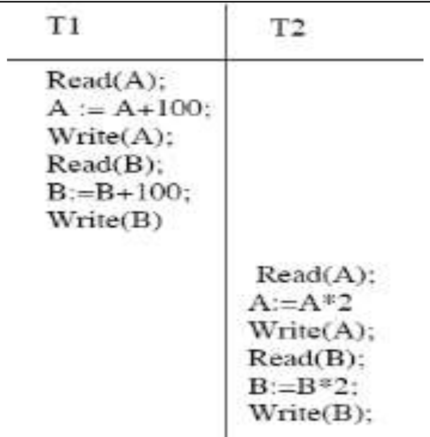
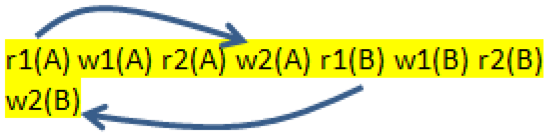
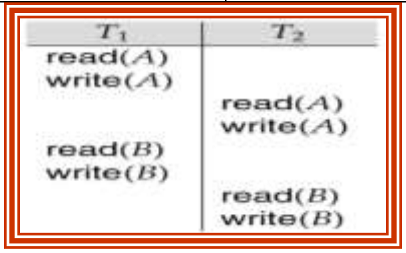
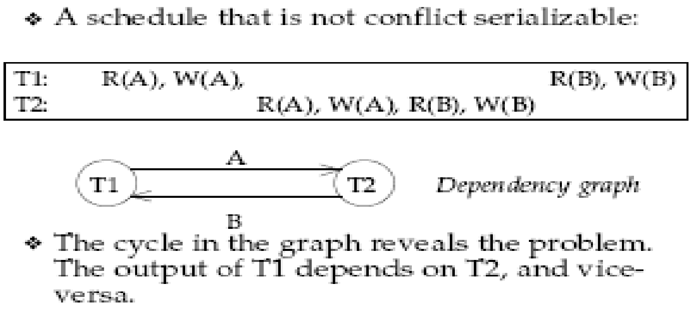
Some example of checking conflict serializable.

We can see that the operations of T1 and T2 in above schedule **C** are interleaved, but we can reorder the actions as shown below so that all the operations of T1 are executed BEFORE the operations of T2.

For check for conflict serializable of Schedule C above, we can create a precedence graph for SCHEDULE C

- Create a node for each transaction
- Add a directed edge from T1 to T2 because T2 reads (Read A, Read B) the value of an item written by T1 (Write A, Write B); T1 executes read before T2 execute write.
- Add a directed edge from T1 to T2 , because T2 writes a value into an item after it has been read by T1. So the graph is acyclic.

In particular, we have the following read and write operations. We denote by r1 a read by transaction T1 and w1 a write by transaction T1. In Schedule C, we have the following read and write operations, in order:

<table>
<tr>
<td>

<table>
<tr><th>T1</th><th>T2</th></tr>
<tr><td>Read(A);
A := A+100;
Write(A);
Read(B);
B:=B+100;
Write(B)</td><td></td></tr>
<tr><td></td><td>Read(A);
A:=A*2
Write(A);
Read(B);
B:=B*2;
Write(B);</td></tr>
</table>
</td>
<td>

r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)
Therefore we can consider the serial schedule show in figure is conflict equivalent to schedule C
</td>
</tr>
<tr>
<td>
<table>
<tr><th><i>T</i>3</th><th><i>T</i>4</th></tr>
<tr><td><b>read</b>(<i>Q</i>)</td><td></td></tr>
<tr><td></td><td><b>write</b>(<i>Q</i>)</td></tr>
<tr><td><b>write</b>(<i>Q</i>)</td><td></td></tr>
</table>
</td>
<td>We are unable to swap instructions in this schedule to obtain either the serial schedule &lt; <i>T</i>3, <i>T</i>4 &gt;, or the serial schedule &lt; <i>T</i>4, <i>T</i>3 &gt;.</td>
</tr>
<tr>
<td>

<table>
<tr><th><i>T</i>1</th><th><i>T</i>2</th></tr>
<tr><td>read(<i>A</i>)
write(<i>A</i>)</td><td></td></tr>
<tr><td></td><td>read(<i>A</i>)
write(<i>A</i>)</td></tr>
<tr><td>read(<i>B</i>)
write(<i>B</i>)</td><td></td></tr>
<tr><td></td><td>read(<i>B</i>)
write(<i>B</i>)</td></tr>
</table>
</td>
<td>This schedule is equivalent to a serial schedule where T2 follows T1, by a series of swaps of non-conflicting instructions. Therefore it is conflict serializable.</td>
</tr>
<tr>
<td></td>
<td>

❖ A schedule that is not conflict serializable:
T1: R(A), W(A), R(B), W(B)
T2: R(A), W(A), R(B), W(B)

❖ The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.
</td>
</tr>
</table>

Recoverability

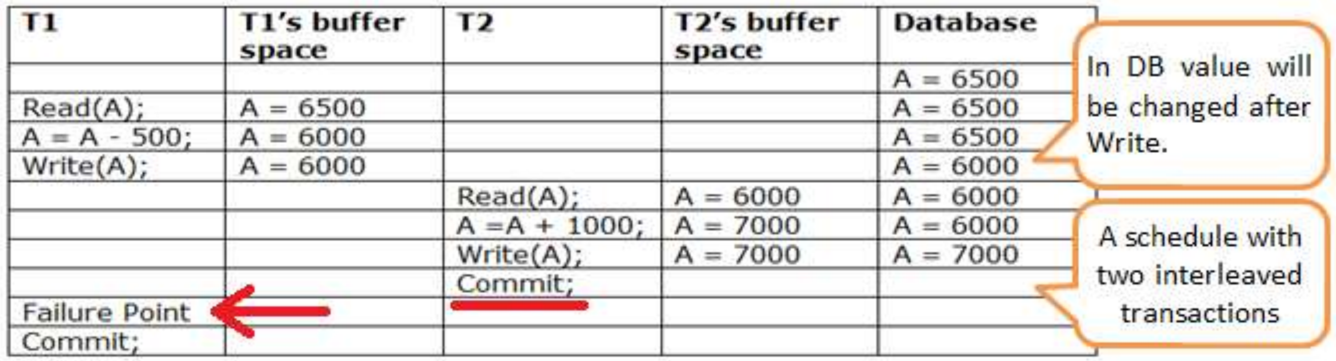
- So far, we have studied what schedules are acceptable from the viewpoint of consistency, assuming implicitly that there are no transaction failures.
- We now address the effect of transaction failures during concurrent execution. We extend our definition of serializability to include aborted transactions.

If a transaction fails, for whatever reason, we need to undo the effect of this transaction. In case of concurrent execution one transaction if aborted we need to abort all dependent transactions. To achieve this surety, we need to place restrictions on the type of schedules permitted in the system.

We will now discuss what schedules are acceptable from the viewpoint of recovery from transaction failure. Most database system requires that all schedules be *recoverable*.

Recoverability of Schedule

A transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.



T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		
Failure Point				
Commit;				

In DB value will be changed after Write.

A schedule with two interleaved transactions

- T1 reads and writes the value of A and that value is read and written by T2.
- T2 commits but later on, T1 fails.
- Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1 (it is dependent on T1), but T2 can't be rollback because it already committed.
- So this type of schedule is known as **irrecoverable schedule**.

Irrecoverable schedule: The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit (see the above schedule).

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
Failure Point				
Commit;				
		Commit;		

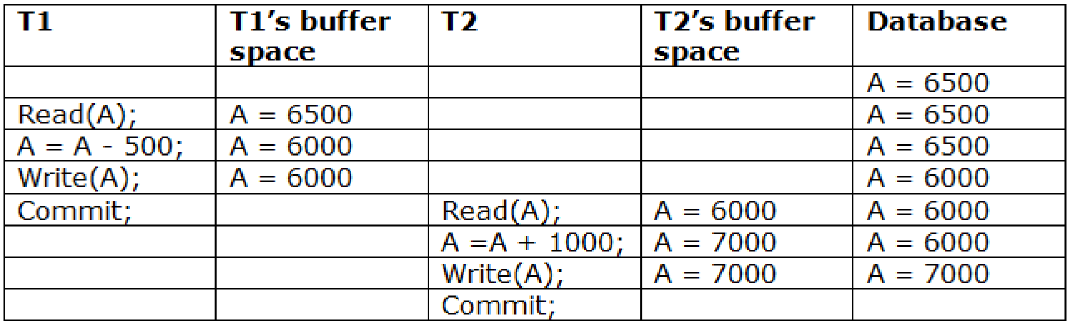
Similar schedule as shown above schedule with two transactions. T2 has read the value written by T1.

Only difference : In T2 commit is done after T1 commit.

- So when T1 fails we have to rollback T1. T2 should be rollback because T2 has read the value written by T1.
- As it has not committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with **cascade rollback**.

Recoverable with cascading rollback: The schedule will be recoverable with cascading rollback if Tj reads the updated value of Ti and Commit of Tj is delayed till commit of Ti.

Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the above schedule where none of the transactions has yet committed (so the schedule is recoverable)



T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
Commit;		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		

The above Table shows a schedule with two transactions. Transaction T1 reads and write A and commits, and that value is read and written by T2. So this is a cascade less recoverable schedule.

In a recoverable schedule, transactions commit only after all transactions whose changes they read commit. If transactions read only the changes of committed transactions, not only is the schedule recoverable, but also aborting a transaction can be accomplished without cascading the abort to other transactions. Such a schedule is said to avoid cascading aborts.

For Implementation of Isolation Schedules must be **serializable**, and **recoverable**, for the sake of database consistency, and preferably cascadeless.

Lock-Based Concurrency Control

A DBMS must be able to ensure following two points:

- Only **serializable**, **recoverable** schedules are allowed
- No actions of **committed transactions are lost while undoing aborted transactions**.

Consider following Example :
Value of **A** changed by T1 from 5 → 6 then by T2 from 6 → 7 then T2 **commits** but T1 **aborts**.
If T1 aborts, the value of A becomes 5 again. Even if T2 commits, its change to A (7) is inadvertently lost. Sometime due to aborting one transaction we may lost the updates of other committed transaction.

A concurrency control technique is needed to solve this problem.

A DBMS typically uses a **concurrency-control protocols** to achieve above two. They allow concurrent schedules, but ensure that the schedules are conflict/view **serializable**, and are **recoverable** and may be even cascadeless.

These protocols do not examine the precedence graph as it is being created, instead a protocol imposes a set of rules that avoids non-seralizable schedules.

Different concurrency control protocols provide different advantages. Different categories of protocols are :

Lock Based Protocol

- Basic 2-PL
- Conservative 2-PL
- Strict 2-PL
- Rigorous 2-PL

2. **Graph Based Protocol**
3. **Time-Stamp Ordering Protocol**
4. **Multiple Granularity Protocol**
5. **Multi-version Protocol**

We will discuss only Lock based protocol

A **locking protocol** is set of rules to be followed by each transaction (and enforced by the DBMS) to ensure that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order.

Lock Based Protocols

- A lock is a variable associated with a data item that describes a status of data item with respect to possible operation that can be applied to it. They synchronize the access by concurrent transactions to the database items.
- This protocol requires that all the data items must be accessed in a mutually exclusive manner.

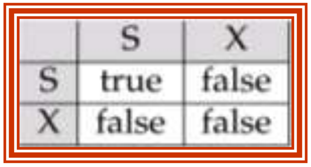
Two common locks which are used in these protocols and how to apply them are discussed below.

1. **Shared Lock (S):** 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.

2. **Exclusive Lock (X):** Data item can be both read as well as written. This is Exclusive and cannot be held simultaneously on the same data item. X-lock is requested using lock-X instruction.

Lock requests are made to **concurrency-control manager or Lock Manager**. Transaction can proceed only after request is granted.

Lock-compatibility matrix



	S	X
S	true	false
X	false	false

Multiple transactions can hold shared locks on the same object. Only one transaction can hold an exclusive lock on an object.

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions (Shared lock is compatible).
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive(X) on the item no other transaction may hold any lock on the item.
- When shared locks (S) allow transactions to read data with SELECT statements. Other transactions are allowed to read the data at the same time; however, no transactions are allowed to modify data until the shared locks are released.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. Then the lock is granted.

Upgrade / Downgrade locks : A transaction that holds a lock on an item **A** is allowed under certain condition to change the lock state from one state to another.

Upgrade: S(A) can be upgraded to X(A) if T_i is the only transaction holding the S-lock on element A. Downgrade: We may downgrade X(A) to S(A) when we feel that we no longer want to write on data-item A. As we were holding X-lock on A, we need not check any conditions.

Applying simple locking may not always produce Serializable results, it may lead to **Deadlock** or **Starvation**.

Problem With Simple Locking…

Consider the Partial Schedule:

	T_1	T_2
1	lock-X(B)	
2	read(B)	
3	B:=B-50	
4	write(B)	
5		lock-S(A)
6		read(A)
7		lock-S(B)
8	lock-X(A)	
9	……	……

Deadlock – T_1 holds an Exclusive lock over B, and T_2 holds a Shared lock over A. Consider Statement 7, T_1 requests for lock on B, while in Statement 8 T_2 requests lock on A.

In this case lock-S(B) requested by T2 cannot be granted until T1 releases the lock-X(B). At the same time T1's request for lock-X(A) cannot be grated until lock-S(A) is released by T2. This imposes a **Deadlock** as none can proceed with their execution.

Starvation – is also possible if concurrency control manager is badly designed. For example: A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item. This may be avoided if the concurrency control manager is good.

Example of Simple locking

T1	**T2**
Lock-X(A)	
Read(A)	
A:= A -100	
Write(A)	
Unlock(A)	
	Lock-S(A)
	Lock-S(B)
	Read(A)
	Read(B)
1900	Display(A+B)
	Unlock(A)
	Unlock(B)
Lock-X(B)	
Read(B)	
B:=B+100	
Write(B)	
Unlock(B)	

Example : Consider a fund transfer between A and B. T1 is transferring fund from A to B. T2 is displaying A+B.

Say initial balance, A= B= 1000. So we should expect A+B = 2000 after the execution of this schedule.

But it will give A+B = **1900**, which is a wrong balance (it should be 2000). The reason behind this anomaly is that T1 has releases the lock on A **too early** in the course of its execution (not holding the lock till the end of transfer).

So Locking as shown is not sufficient to guarantee serializability. Note that the transactions unlock a data item immediately after its final access is not always desirable, since serializability may not be ensured.

If we create Transaction T3 corresponds to T1 with unlocking delayed and Transaction T4 corresponds to T2 with unlocking delayed then we will be able to avoid this wrong display with

	the same schedule. In two phase locking scheme (protocol) such premature release of a lock will not be permitted.

Simple Lock based protocol (or *Binary Locking*) implements this simple lock system without any restrictions. It has its own disadvantages, they **does not guarantee Serializability**.

We have to apply Locks but they must follow a set of protocols to avoid such undesirable problems. Shortly we'll use 2-Phase Locking (2-PL) which will use the concept of Locks to avoid deadlock.

To guarantee serializablity, 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 serializablity**.

Why Locking is not adequate in its own to guarantee serializability?

Following example demonstrate this.

	T1	T2	
	Lock-S(B)		
B=30	R(B)		
	Unlock(B)		
		Lock-S(A)	
		R(A)	A=20
		Unlock(A)	
		Lock-X(B)	
		R(B)	B=30
		B = A+ B	B=20+30=50
		W(B)	
		Unlock(B)	
	Lock-X(A)		
A=20	R(A)		
A=20	A = A+ B		
+50	W(A)		
=70	Unlock(A)		

Initial Values : A=20; B=30

- Serial Schedule T1→T2 gives result **A=50, B = 80**
- Serial Schedule T2→T1 gives result **A=70, B = 50**
- Result of interleaved schedule on left **A=70, B=50**

Since the result is not equal to any of the serial schedule the left schedule is **Nonserializabel**.

This example demonstrates that locks cannot be obtained in an arbitrary manner.

Two Phase Locking

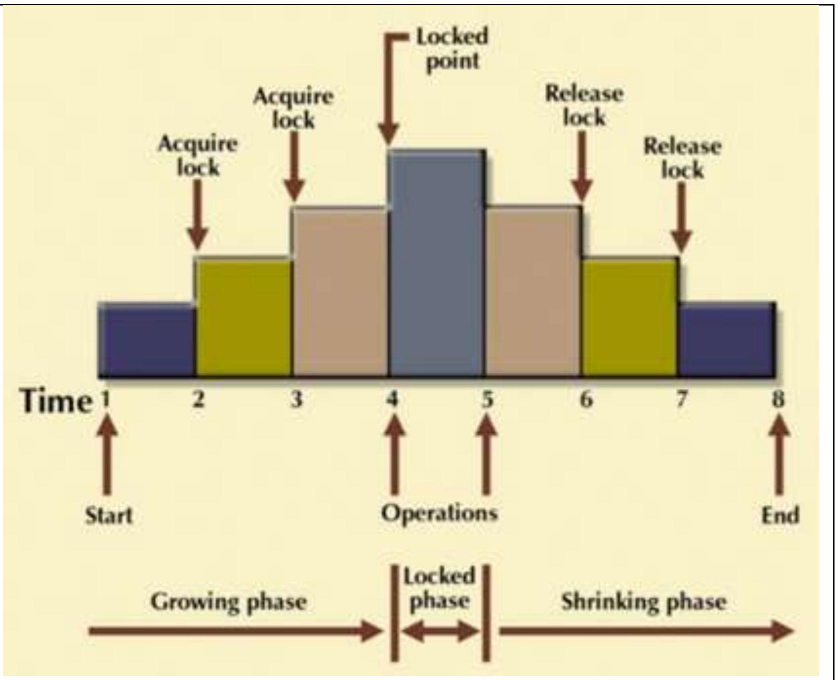
The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two phases:

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.

It is true that the 2PL protocol offers serializability. However, it does not ensure that deadlocks do not happen.

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.



If every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable.

Let's see a transaction implementing 2-PL.

	T_1	T_2
1	**LOCK-S(A)**	
2		**LOCK-S(A)**
3	**LOCK-X(B)**	
4	**…….**	**……**
5	**UNLOCK(A)**	
6		**LOCK-X(C)**
7	**UNLOCK(B)**	
8		**UNLOCK(A)**
9		**UNLOCK(C)**
10	**…….**	**……**

This is just a skeleton transaction which shows how unlocking and locking works with 2-PL.

Transaction T_1:

- Growing Phase is from steps 1-3.
- Shrinking Phase is from steps 5-7.
- Lock Point at 3

Transaction T_2:

- Growing Phase is from steps 2-6.
- Shrinking Phase is from steps 8-9.
- Lock Point at 6

What is **LOCK POINT ?** The Point at which the growing phase ends, i.e., when transaction takes the final lock it needs to carry on its work.

Now, let’s see the schedule below, we are applying the 2-PL.

Example of 2PL

	T_1	T_2
1	Read(A)	
2		Read(A)
3	Read(B)	
4	Write(B)	
5	**Commit**	
6		Read(B)
7		Write(B)
6		**Commit**

Let’s see the schedule above which is conflict serializable. So we can try implementing 2-PL, which is shown in adjacent table. Note that here locks are released after Commit operation so this satisfies Rigorous 2-PL protocol.

	T_1	T_2
1	Lock-S(A)	
2	Read(A)	
3		Lock-S(A)
4		Read(A)
5	Lock-X(B)	
6	Read(B)	
7	Write(B)	
8	**Commit**	
9	Unlock(A)	
10	Unlock(B)	
11		Lock-X(B)
12		Read(B)
13		Write(B)
14		**Commit**
15		Unlock(A)
16		Unlock(B)

In growing phase after acquiring a lock permitted operation can be done but lock will not be released.

Now, this is one way I choose to implement the locks on A and B. You may try a different sequence but remember to follow the 2-PL protocol.

Another example

T1	**T2**	

Lock-X(A)
Read(A)
A:= A -100
Write(A)
Unlock(A)
		In this example T1 does not satisfy two phase requirement. It has released the lock on A and then has requested for lock on B.
		But T2 satisfies the two phase requirement. It has not requested for another lock after releasing any lock.

	Lock-S(A)
	Read(A)
	Lock-S(B)
	Read(B)
	Display(A+B)
	Unlock(A)
	Unlock(B)	

Lock-X(B)
Read(B)
B:=B+100
Write(B)
Unlock(B)		

In basic two phase locking protocol :

- Once a lock is granted to a transaction, it is not unlocked till the end of processing
- Once a lock is released for a transaction, the same transaction is not allowed to get any more locks.

2-PL ensures serializablity, but there are still some drawbacks of 2-PL. The drawbacks are :

- Cascading Rollback is possible under 2-PL.
- Deadlocks and Starvation is possible.
- Two phase locking may also limit the **amount of concurrency** that occurs in a schedule because a Transaction may not be able to release an item after it has used it. This is the price we have to pay to ensure serializablity and other factors, hence it can be considered as a bargain between concurrency and maintaining the ACID properties.

The above mentioned type of 2-PL is called **Basic 2PL**.

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

Rules used in Strict 2-PL

This requires that in addition to the lock being 2-Phase **all Exclusive(X) Locks** held by the transaction be released *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*!

Rules used in 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 ***after* the Transaction Commits**.
Following Rigorous 2-PL ensures that our schedule is:

- Recoverable
- Cascadeless

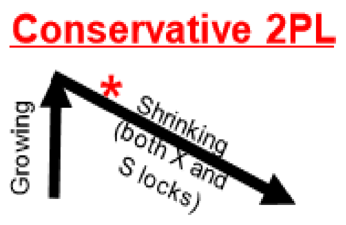
Hence it gives us freedom from Cascading Abort which was still there in Basic 2-PL and moreover guarantee Strict Schedules but still *Deadlocks are possible*!

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 more easy.

Rules used in Conservative 2-PL –

It is also known as **Static 2-PL**, this protocol requires :

- Locks all desired data items before 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*. However, it is difficult to use in practice because of 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.

Guarantees Serializability but Not Strict Schedule for Recoverability, as some transaction can make a dirty read from T, before T has committed

Deadlock in DBMS

In a database, a deadlock is an unwanted situation in which two or more transactions are waiting indefinitely for one another to give up locks. Deadlock is said to be one of the most feared complications in DBMS as it brings the whole system to a Halt.

Example – let us understand the concept of Deadlock with an example :

Suppose, Transaction T1 holds a lock on some rows in the **Students** table and needs to update some rows in the **Grades** table. Simultaneously, Transaction T2 holds locks on those very rows (Which T1 needs to update) in the **Grades** table but needs to update the rows in the Student table held by Transaction T1.

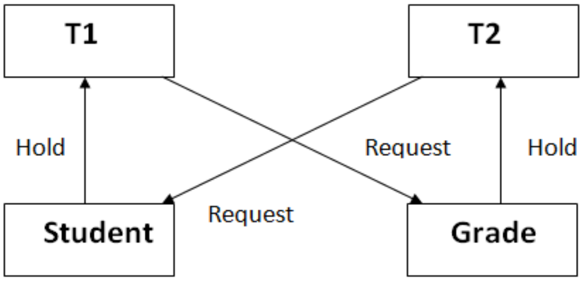


Figure: Deadlock in DBMS

Now, the main problem arises. Transaction T1 will wait for transaction T2 to give up lock, and similarly transaction T2 will wait for transaction T1 to give up lock. As a consequence, all activity comes to a halt and remains at a standstill forever unless the DBMS detects the deadlock and aborts one of the transactions.

Deadlock Avoidance

- It is better to avoid the deadlock rather than aborting or restating the database. This is a waste of time and resource. Deadlock avoidance method is suitable for smaller database.

- Deadlock avoidance mechanism is used to detect any deadlock situation in advance. A method like "**wait for graph**" is used for detecting the deadlock situation but this method is suitable only for the smaller database. For the larger database, deadlock prevention method can be used.

Deadlock Detection

In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not. The lock manager maintains a Wait for the graph to detect the deadlock cycle in the database.

Deadlock prevention

For large database, deadlock prevention method is suitable. A deadlock can be prevented if the resources are allocated in such a way that deadlock never occur. The DBMS analyzes the operations whether they can create deadlock situation or not, If they do, that transaction is never allowed to be executed.

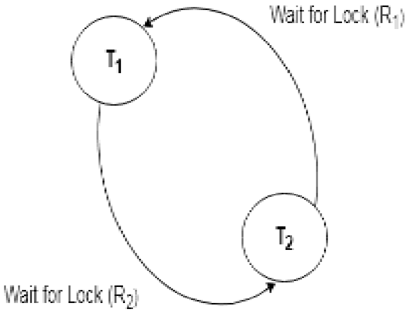
Wait-for graph

- A wait-for graph in computer science is a directed graph used for deadlock detection in operating systems and RDBMS.

This method for deadlock detection is suitable for smaller database. In this method a graph is drawn based on the transaction and their lock on the resource. If the graph created has a closed loop or a cycle, then there is a deadlock.

The wait for the graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph.

For the above mentioned scenario the Wait-For graph is drawn as shown in figure.



Detecting deadlocks using wait-for graphs

Wait-for graph is DBMS is a graph where :

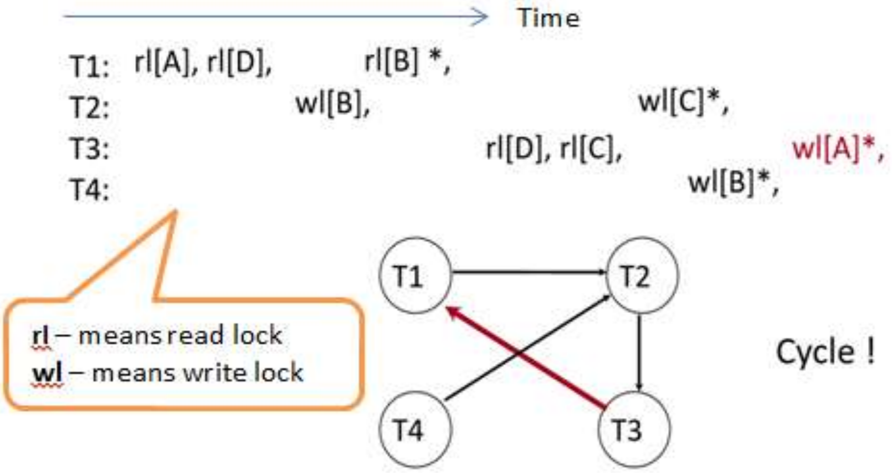
- Node represents **transaction**
- Edge **i => j** represents the fact that
  "the transaction i is waiting for a lock held by the transaction j".

- The **wait-for** graph can be **constructed** using the **information** stored in the **lock table** entries

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j
  - implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item held by T_j, then $T_i \rightarrow T_j$ is inserted in the wait-for graph.
  - This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- The system invokes a deadlock-detection algorithm periodically to look for cycles.

Example of deadlock detection

For simplicity, the lock requests only are shown, and those with * are suspended



Two schemes of Deadlock prevention

Wait-Die Scheme

In this scheme, if a transaction request for a resource that is locked by other transaction, then the DBMS simply checks the timestamp of both transactions and allows the older transaction to wait until the resource is available for execution.

Let's assume there are two transactions Ti and Tj and let TS(T) is a timestamp of any transaction T. If T2 holds a lock by some other transaction and T1 is requesting for resources held by T2 then the following actions are performed by DBMS:

1. Check if TS(Ti) < TS(Tj) - If Ti is the older transaction and Tj has held some resource, then Ti is allowed to wait until the data-item is available for execution. That means if the older transaction is

waiting for a resource which is locked by the younger transaction, then the older transaction is allowed to wait for resource until it is available.

2. Check if TS(T_i) < TS(Tj) - If Ti is older transaction and has held some resource and if Tj is waiting for it, then Tj is killed and restarted later with the random delay but with the same timestamp.

```
IF ts(Ti) < ts(Tj)            (Ti is older than  Tj)
     THEN Ti wait for Tj     (older waits for the younger)
ELSE Ti aborts                (younger dies)
If Ti dies then it later restarts with the same timestamp!
```

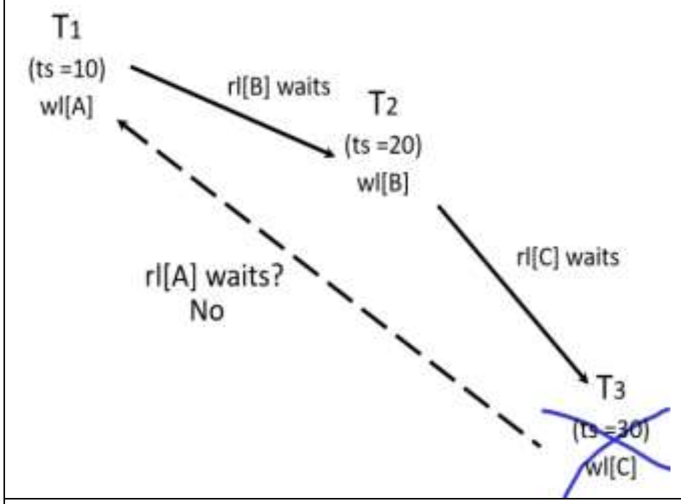
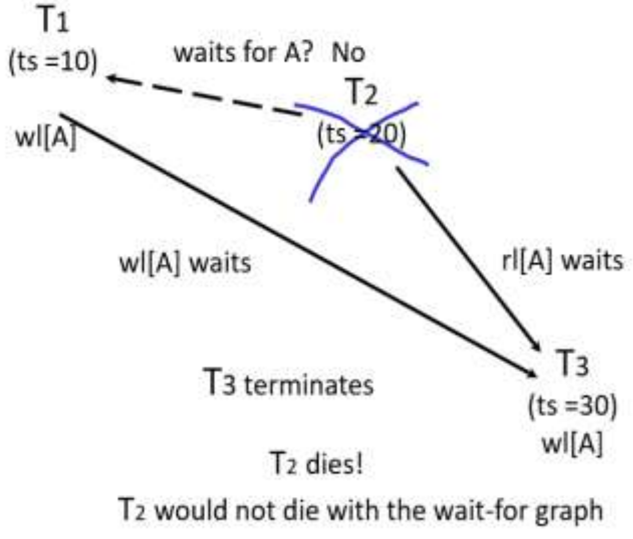
Wound wait scheme

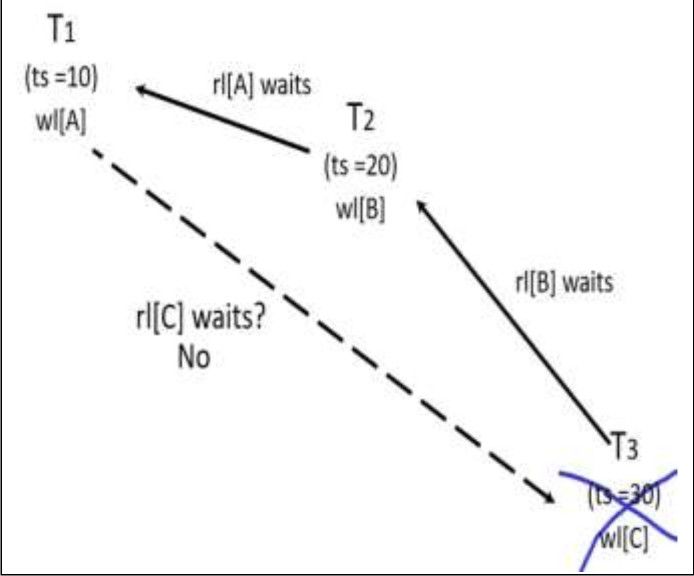
- In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then older transaction forces younger one to kill the transaction and release the resource. After the minute delay, the younger transaction is restarted but with the same timestamp.
- If the older transaction has held a resource which is requested by the Younger transaction, then the younger transaction is asked to wait until older releases it.

```
IF ts(Ti) < ts(Tj)                                  (Ti is older than Tj)
    THEN Ti wounds Tj and takes the lock    (younger dies: lock to older)
ELSE Ti waits                                       (younger waits for older)

If Tj dies then it later restarts with the same timestamp
```

Examples of Deadlock prevention

Wait-die: example  T1 (ts =10) wl[A] rl[B] waits T2 (ts =20) wl[B] rl[C] waits rl[A] waits? No T3 (ts =30) wl[C]	 T1 (ts =10) waits for A? No T2 (ts =20) wl[A] wl[A] waits rl[A] waits T3 (ts =30) wl[A] T3 terminates T2 dies! T2 would not die with the wait-for graph
Wound-wait : example	**Comparing Deadlock Management Schemes** • Wait-die and Wound-wait ensure no starvation



- Wait-die (older waits) tends to roll back more transactions then Wound-wait (younger waits) but they tend to have done less work
- Wait-die and Wound-wait are easier to implement than waits-for graph
- Waits-for graph technique only aborts transactions if there really is a deadlock (unlike the others)