Transactions

Transaction Concept, A Simple Transaction Model, Storage Structure, Transaction Atomicity and Durability, Transaction Isolation, Serializability, Transaction Isolation and Atomicity.

Concurrency Control

Lock-Based Protocols, Deadlock Handling, Timestamp- Based Protocols, validation based protocols.

Transaction Concept

In **Database Management Systems (DBMS)**, a transaction is a fundamental concept representing a set of logically related operations executed as a **single unit**.

Transactions are essential for handling user requests to access and modify database contents, ensuring the database remains consistent and reliable despite various operations and potential interruptions.

Example: Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:

X's Account

```
Open_Account(X)
Old_Balance = X.balance
New_Balance = Old_Balance - 800
X.balance = New_Balance
Close_Account(X)
```

Y's Account

```
Open_Account(Y)
Old_Balance = Y.balance
New_Balance = Old_Balance + 800
Y.balance = New_Balance
Close_Account(Y)
```

A Simple Transaction Model

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

Operations of Transaction:

Following are the main operations of transaction:

Read(X): Read operation is used to read the value of X from the database and stores it in a buffer in main memory.

Write(X): Write operation is used to write the value back to the database from the buffer.

Let's take an example to debit transaction from an account which consists of following operations:

```
R(X);
X = X - 500;
W(X);
```

Let's assume the value of X before starting of the transaction is 4000.

- The first operation reads X's value from database and stores it in a buffer.
- The second operation will decrease the value of X by 500. So buffer will contain 3500.
- The third operation will write the buffer's value to the database. So X's final value will be 3500.

But it may be possible that because of the failure of hardware, software or power, etc. that transaction may fail before finished all the operations in the set.

For example: If in the above transaction, the debit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the bank.

To solve this problem, we have two important operations:

Commit: It is used to save the work done permanently.

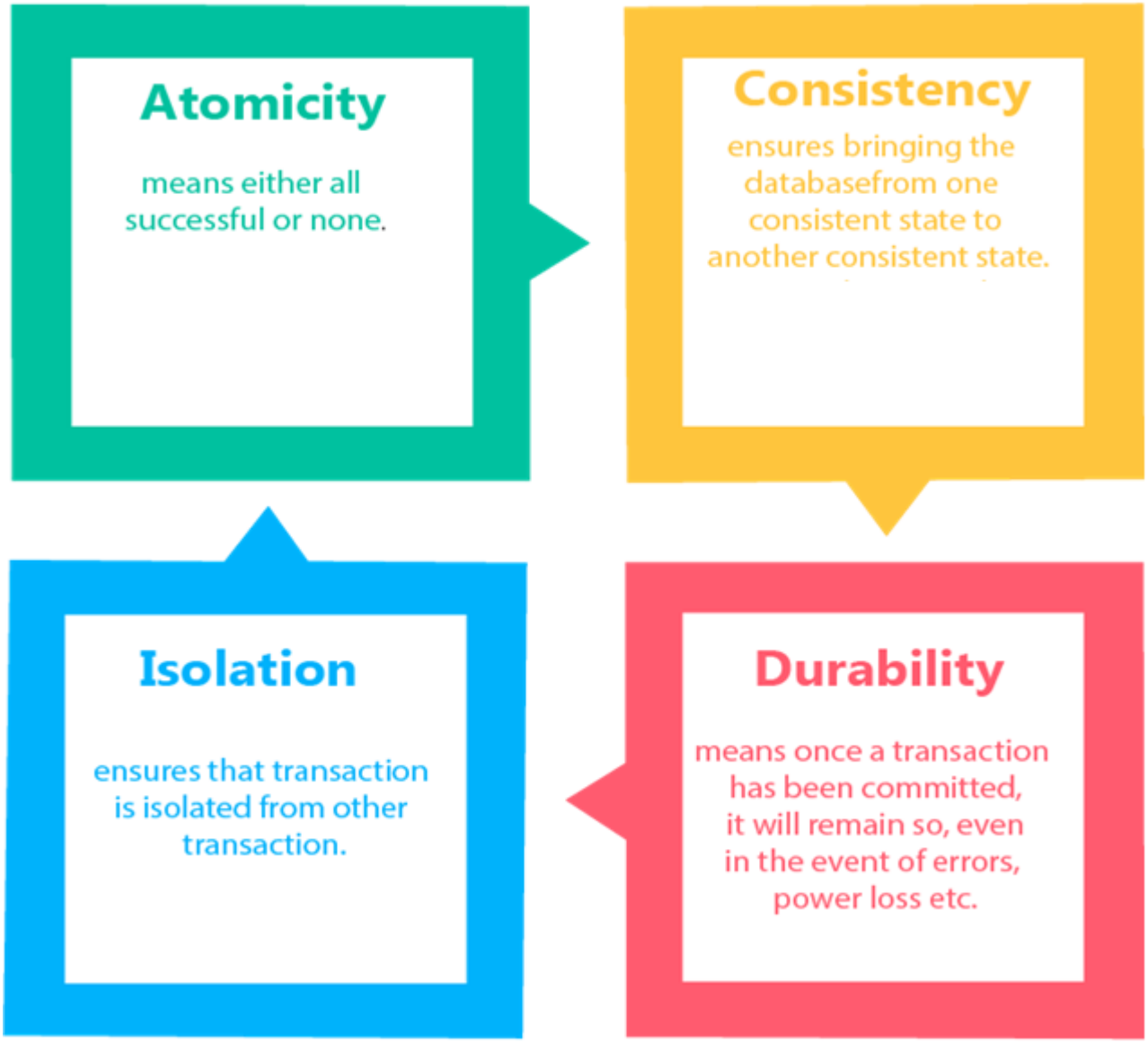
Rollback: It is used to undo the work done.

Transaction Properties

The transaction has the four properties. These are used to maintain consistency in a database, before and after the transaction.

Properties of Transaction

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



Atomicity

- It states that all operations of the transaction take place at once if not, the transaction is aborted.
- There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.

Atomicity involves the following two operations:

Commit: If a transaction commits then all the changes made are visible.

Abort: If a transaction aborts then all the changes made are not visible.

Example: Let's assume that following transaction T consisting of T1 and T2. A consists of Rs 600 and B consists of Rs 300. Transfer Rs 100 from account A to account B.

T1	T2
Read(A) A:= A-100 Write(A)	Read(B) B:= B+100 Write(B)

After completion of the transaction, A consists of Rs 500 and B consists of Rs 400.
If the transaction T fails after the completion of transaction T1 but before completion of transaction T2, then the amount will be deducted from A but not added to B. This shows the inconsistent database state. In order to ensure correctness of database state, the transaction must be executed in entirety.

Consistency

- The integrity constraints are maintained so that the database is consistent before and after the transaction.
- The execution of a transaction will leave a database in either its prior stable state or a new stable state.
- The consistent property of database states that every transaction sees a consistent database instance.
- The transaction is used to transform the database from one consistent state to another consistent state.

For example: The total amount must be maintained before or after the transaction.

Total before T1 occurs = 600+300=900
Total after T2 occurs= 500+400=900

Therefore, the database is consistent. In the case when T1 is completed but T2 fails, then inconsistency will occur.

Isolation

- It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
- In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1 ends.
- The concurrency control subsystem of the DBMS enforced the isolation property.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

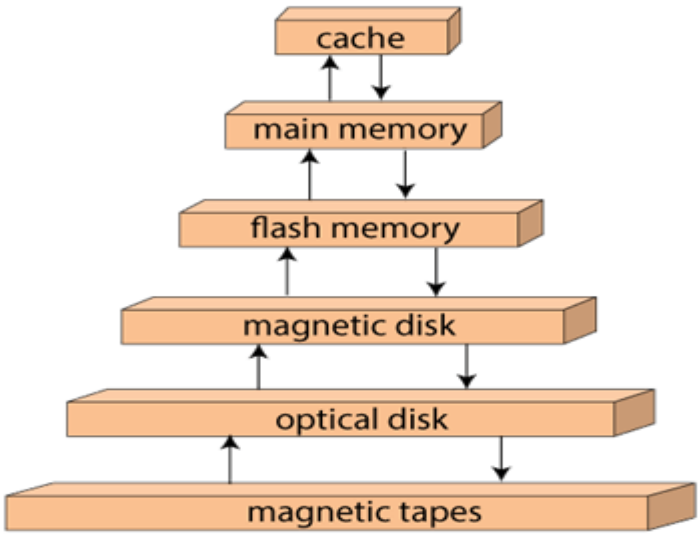
Schedule 1 — a serial schedule in which T_1 is followed by T_2.

Durability

- Once the DBMS informs the user that a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk. This property is called durability
- The durability property is used to indicate the performance of the database's consistent state. It states that the transaction made the permanent changes.
- They cannot be lost by the erroneous operation of a faulty transaction or by the system failure. When a transaction is completed, then the database reaches a state known as the consistent state. That consistent state cannot be lost, even in the event of a system's failure.
- The recovery subsystem of the DBMS has the responsibility of Durability property.

Storage Structure

To understand how to ensure the atomicity and durability properties of a transaction, we must gain a better understanding of how the various data items in the database may be stored and accessed.



Storage device hierarchy

Types of Data Storage

For storing the data, there are different types of storage options available. These storage types differ from one another as per the speed and accessibility. There are the following types of storage devices used for storing the data:

- **Primary Storage**
- **Secondary Storage**
- **Tertiary Storage**

Storage media can be distinguished by their relative speed, capacity, and resilience to failure, and classified as **volatile** storage or **nonvolatile** storage or **stable storage**.

1. **Volatile Storage**: Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself, and because it is possible to access any data item in volatile storage directly.
2. **Nonvolatile Storage**: Information residing in nonvolatile storage survives system crashes. Examples of nonvolatile storage include secondary storage devices such as magnetic disk and flash storage, used for online storage, and tertiary storage devices such as optical media, and magnetic tapes, used for archival storage. At the current state of technology, nonvolatile storage is slower than volatile storage, particularly for random access. Both secondary and tertiary storage devices, however, are susceptible to failure which may result in loss of information.
3. **Stable Storage** Information residing in stable storage is *never* lost. To implement stable storage, we replicate the information in several nonvolatile storage media (usually disk) with independent failure modes. Updates must be done with care to ensure that a failure during an update to stable storage does not cause a loss of information.

For example, some RAID controllers, provide battery backup, so that some main memory can survive system crashes and power failures.

For a transaction to be durable, its changes need to be written to stable storage. Similarly, for a transaction to be atomic, log records need to be written to stable storage before any changes are made to the database on disk. Clearly, the degree to which a system ensures durability and atomicity depends on how stable its implementation of stable storage really is. In some cases, a single copy on disk is considered sufficient, but applications whose data are highly valuable and whose transactions are highly important require multiple copies, or, in other words, a closer approximation of the idealized concept of stable storage.

Transaction Atomicity and Durability

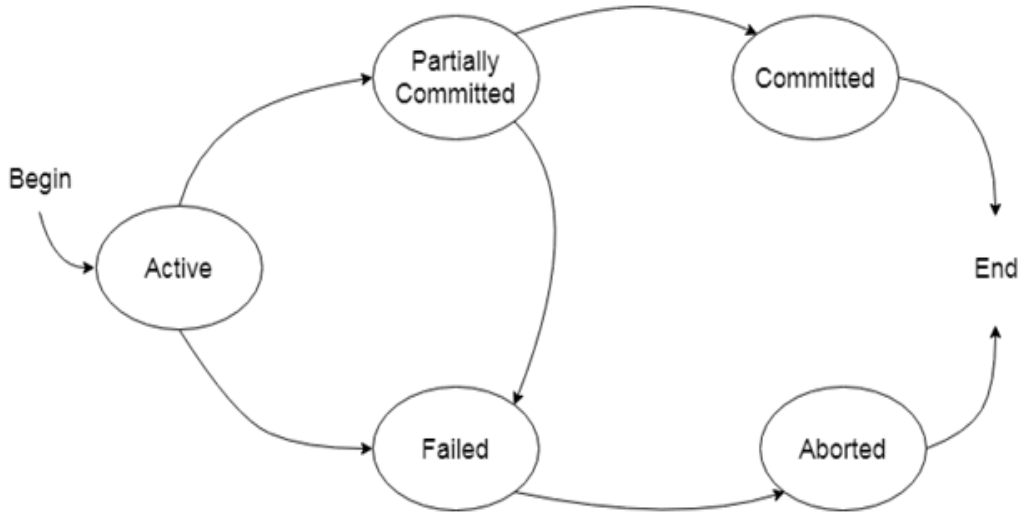
Atomicity and durability are ACID properties that ensure the reliability and consistency of transactions in DBMS. First of all, let's understand Atomicity and Durability.

Atomicity: By this property, we mean that the entire transaction takes place at once or it does not take place at all. There is no midway. This means that transactions do not occur partially. Either all the changes made by a transaction are committed to the database or none of them are committed.

Durability: By this property, we mean that once a transaction has completed execution its effects persist even in case of system failure. The changes made by the transaction should be stored permanently in the system.

States of Transaction

In a database, the transaction can be in one of the following states -



Active State

- The active state is the first state of every transaction. In this state, the transaction is being executed.
- For example: Insertion or deletion or updating a record is done here. But all the records are still not saved to the database.

Partially Committed

- In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database.
- In the total mark calculation example, a final display of the total marks step is executed in this state.

Committed

A transaction is said to be in a committed state if it executes all its operations successfully. In this state, all the effects are now permanently saved on the database system.

Failed State

1. If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
2. In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

Aborted

- If any of the checks fail and the transaction has reached a failed state then the database recovery system will make sure that the database is in its previous consistent state. If not then it will abort or roll back the transaction to bring the database into a consistent state.
- If the transaction fails in the middle of the transaction then before executing the transaction, all the executed transactions are rolled back to its consistent state.
- After aborting the transaction, the database recovery module will select one of the two operations:
    - **Re-start the transaction**
    - **Kill the transaction**

Transaction Isolation

Transaction-Processing Systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data, as we saw earlier.

Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run **serially**—that is, one at a time, each starting only after the previous one has completed. However, there are two good reasons for allowing concurrency:

- **Improved throughput and resource utilization**. A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the **throughput** of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk **utilization** also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.
- **Reduced Waiting Time**. There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running transactions.

Consider again the simplified banking system, which has several accounts and a set of transactions, that access and update those accounts.

Let $T1$ and $T2$ be two transactions that transfer funds from one account to another.

Transaction *T*1 transfers $50 from account *A* to account *B*. It is defined as:

***T*1:** read(*A*);
A := *A* − 50;
write(*A*);
read(*B*);
B := *B* + 50;
write(*B*).

Transaction *T*2 transfers 10 percent of the balance from account *A* to account *B*.

***T*2:** read(*A*);
temp := *A* * 0.1;
A := *A* − *temp*;
write(*A*);
read(*B*);
B := *B* + *temp*;
write(*B*).

Suppose the current values of accounts *A* and *B* are $1000 and $2000, respectively.

Suppose also that the two transactions are executed one at a time in the order *T*1 followed by *T*2. This execution sequence appears in Figure 14.2. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of *T*1 appearing in the left column and instructions of *T*2 appearing in the right column. The final values of accounts *A* and *B*, after the execution takes place, are $855 and $2145, respectively. Thus, the total amount of money in accounts *A* and *B*—that is, the sum *A* + *B*—is preserved after the execution of both transactions.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Schedule 1 — a serial schedule in which T_1 is followed by T_2.

The execution sequences just described are called **schedules**. They represent the chronological order in which instructions are executed in the system. Clearly, a schedule for a set of transactions must consist of all instructions of those transactions, and must preserve the order in which the instructions appear in each individual transaction.

These schedules are **serial**: Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.

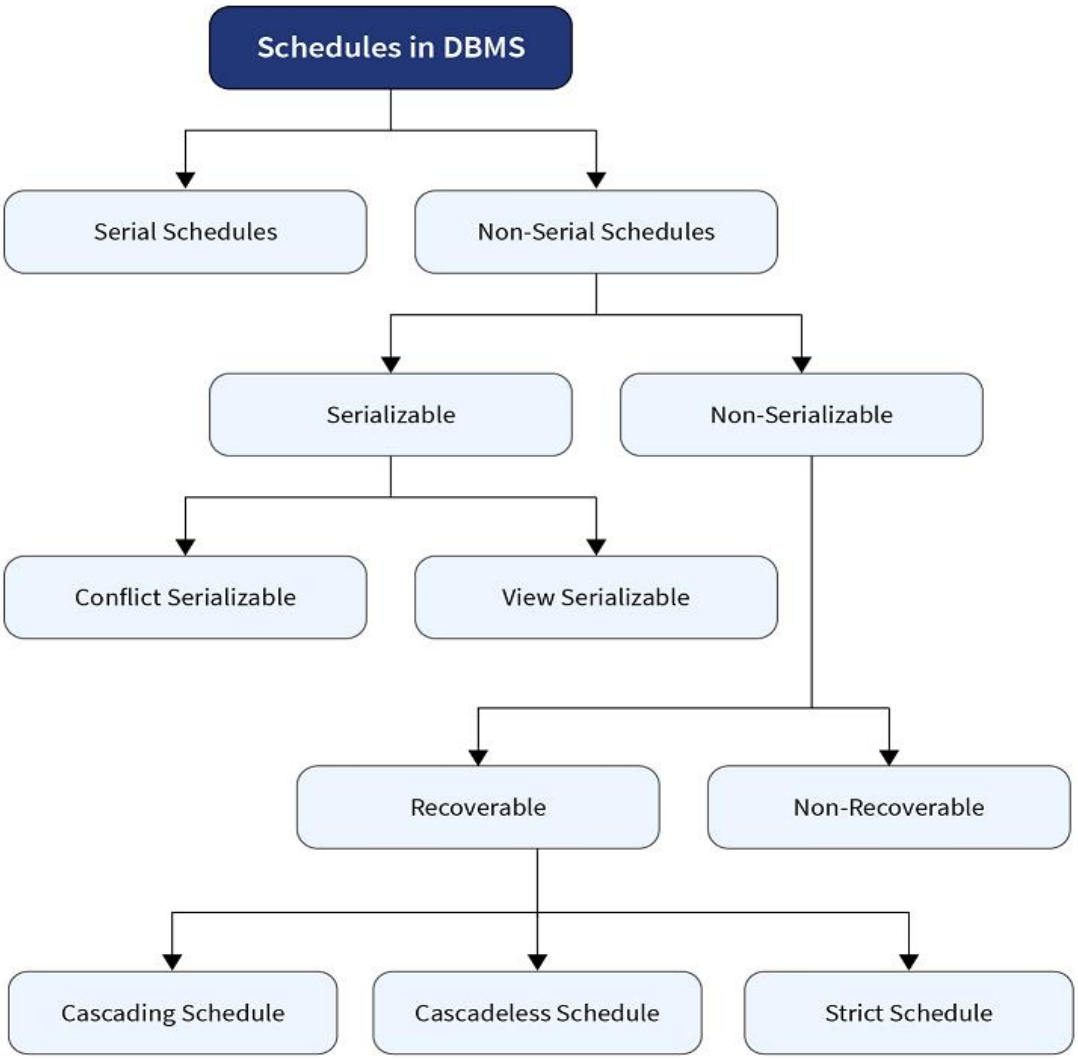
Serializability

Serializability in DBMS guarantees that the execution of multiple transactions in parallel does not produce any unexpected or incorrect results. This is accomplished by enforcing a set of rules that ensure that each transaction is executed as if it were the only transaction running in the system.

Schedules in DBMS

A schedule refers to the chronological order of executing transactions in a multi-user environment. A schedule defines the sequence in which various transactions read and write data items in the database.

A schedule can include multiple transactions running concurrently; each performing a series of read and write operations on the database.



1. **Serial Schedules**

The serial schedule is a type of schedule where each transaction is executed in its entirety before the next transaction begins. This schedule can be viewed as the simplest and most straightforward type of schedule, as it guarantees that transactions are executed in isolation from each other.

Example: Consider the following schedule involving two transactions T 1 and T 2 .

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

where R(A) denotes that a read operation is performed on some data item 'A' This is a serial schedule since the transactions perform serially in the order T 1 —> T 2

2. **Non-Serial Schedule in DBMS:**

The Non-Serial schedule is a type of schedule where transactions are executed concurrently, with some overlap in time. Unlike a serial schedule, where transactions are executed one after the other with no overlap, a non-serial schedule allows transactions to execute simultaneously.

Example of Non-Serial Schedule:

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

What is a Serializable Schedule?

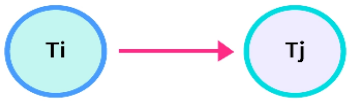
A non-serial schedule is called a serializable schedule if it can be converted to its equivalent serial schedule.

Testing of Serializability

To test the serializability of a schedule, we can use Serialization Graph or Precedence Graph. A serialization Graph is nothing but a Directed Graph of the entire transactions of a schedule.

It can be defined as a Graph G(V, E) consisting of a set of directed-edges E = {E1, E2, E3, ..., En} and a set of vertices V = {V1, V2, V3, ...,Vn}. The set of edges contains one of the two operations - READ, WRITE performed by a certain transaction.
READ, WRITE performed by a certain transaction.

Precedence Graph for schedule S



Ti -> Tj, means Transaction-Ti is either performing read or write before the transaction-Tj.

Types of Serializability

There are mainly two types of serializability:

1. View Serializability
2. Conflict Serializability

1. View Serializability in DBMS:

View Serializability is the process of determining whether or not a given schedule is view serializable. If a schedule is a view equivalent to a serial schedule, it is view serializable.

To test for view serializability, we first identify the read and write operations of each transaction. A schedule is considered view serializable if it is view equivalent to a serial schedule, which is a schedule where the transactions are executed one after the other without any overlap.

Example of View Serializability in DBMS:

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

Let's swap the read-write operations in the middle of the two transactions to create the view equivalent schedule.

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

2. Conflict Serializability :

A schedule's ability to prevent the sequence of conflicting transactions from having an impact on the transactions' results is known as conflict serializability in DBMS. Conflicting transactions are those that make unauthorized changes to the same database data item.

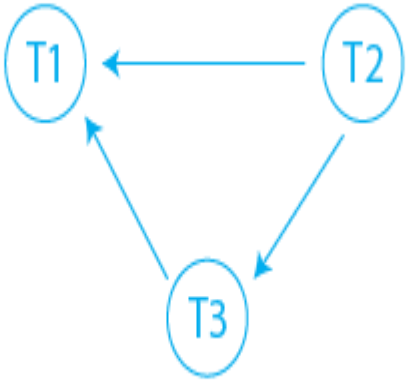
Key conditions for conflict serializability.

- Both operations belong to different transactions
- Both operations are on the same data item
- At least one of the two operations is a write operation

Example of Conflict Serialzability :

Transaction 1	Transaction 2	Transaction 3
R(X)		
		R(Y)
		R(X)
	R(Y)	
	R(Z)	
		W(Y)
	W(Z)	
R(Z)		
W(X)		
W(Z)		

Let's see how a precedence graph looks like of above schedule.



In the above precedence graph, we can see that there is no cycle present in the graph. So, we can say that the above schedule is **Conflict Serializable.**

Transaction Isolation and Atomicity

Now we address the effect of transaction failures during concurrent execution.

Non-Serializable:

The non-serializable schedule is divided into two types,

1. Recoverable
2. Non-recoverable Schedule.

1. **Recoverable Schedule:**

Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction T j is reading value updated or written by some other transaction T i , then the commit of T j must occur after the commit of T i .

Example – Consider the following schedule involving two transactions T 1 and T 2 .

T1	T2
R(A)	
W(A)	
	W(A)
	R(A)
commit	
	commit

This is a recoverable schedule since T 1 commits before T 2 , that makes the value read by T 2 correct.

There can be three types of recoverable schedule:

1. Cascading Schedule
2. Cascadeless Schedule
3. Strict Schedule

1. Cascading Schedule:

Also called Avoids cascading aborts/rollbacks (ACA). When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or cascading abort. Example:

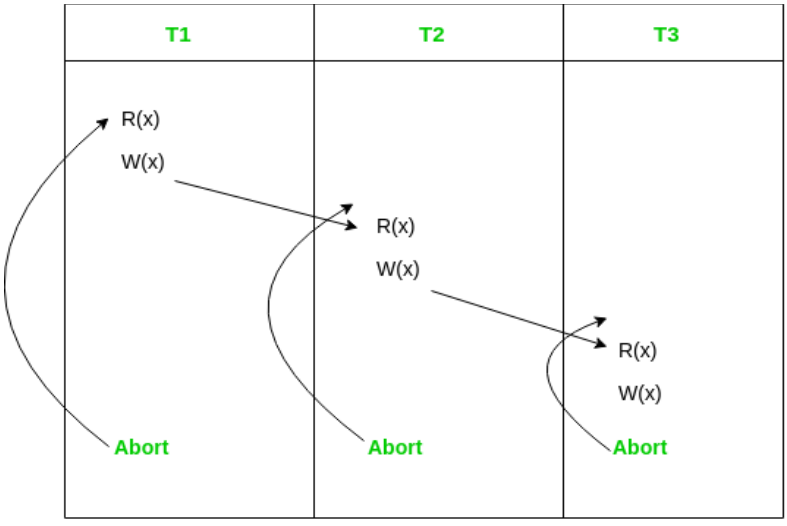


Figure - Cascading Abort

2. **Cascadeless Schedule:**

Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule. In other words, if some transaction T j wants to read value updated or written by some other transaction T i , then the commit of T j must read it after the commit of T i .

Example: Consider the following schedule involving two transactions T 1 and T 2 .

T1	T2
R(A)	
W(A)	
	W(A)
commit	
	R(A)
	commit

This schedule is cascadeless. Since the updated value of **A** is read by T 2 only after the updating transaction i.e. T 1 commits.

Example: Consider the following schedule involving two transactions T 1 and T 2 .

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
abort	
	abort

It is a recoverable schedule but it does not avoid cascading aborts. It can be seen that if T 1 aborts, T 2 will have to be aborted too in order to maintain the correctness of the schedule as T 2 has already read the uncommitted value written by T 1 .

3. **Strict Schedule:**

A schedule is strict if for any two transactions T i , T j , if a write operation of T i precedes a conflicting operation of T j (either read or write), then the commit or abort event of T i also precedes that conflicting operation of T j . In other words, T j can read or write updated or written value of T_i only after T_i commits/aborts. **Example:** Consider the following schedule involving two transactions T_1 and T_2 .

T1	T2
R(A)	
	R(A)
W(A)	
commit	
	W(A)
	R(A)
	commit

This is a strict schedule since T 2 reads and writes A which is written by T 1 only after the commit of T 1 .

2. **Non-Recoverable Schedule:**

A non-recoverable schedule is a schedule in DBMS that prevents the database from recovering to a consistent state if a system failure occurs. This happens when the commit operation of transaction Ti does not occur before the commit operation of transaction Tj.

Example: Consider the following schedule involving two transactions T 1 and T 2 .

T1	T2
R(A)	
W(A)	
	W(A)
	R(A)
	commit
abort	

T 2 read the value of A written by T 1 , and committed. T 1 later aborted, therefore the value read by T 2 is wrong, but since T 2 committed, this schedule is **non-recoverable** .

Concurrency Control

Lock-Based Protocols, Deadlock Handling, Timestamp- Based Protocols, validation based protocols.

Concurrency Control

Concurrency Control is a process of managing and executing simultaneous transactions or manipulation of data by multiple processes or by users without data loss, data integrity, and data consistency.

In a multi-user database environment, where various users are accessing and modifying the database and executing or manipulating the transactions, it is necessary to control the concurrency.

Thus, for maintaining the concurrency of the database, we have the concurrency control protocols.

Concurrency Control Protocols

The concurrency control protocols ensure the *atomicity, consistency, isolation, durability* and *serializability* of the concurrent execution of the database transactions. Therefore, these protocols are categorized as:

1. Lock Based Protocols
2. Time Stamp Based Protocols
3. Validation Based Protocols

1. Lock-Based Protocols

One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a **lock** on that item.

1. Simple Lock Based Protocol
2. Two-Phase Locking (2PL) Protocol
    a) Rigorous Two-Phase Locking
    b) Strict Two-Phase Locking

1. Simple Lock Based Protocol

The Simple lock based protocol is a mechanism in which there is exclusive use of locks on the data item for current transaction.

Types of Locks: There are two types of locks used –

1. **Shared**. If a transaction *Ti* has obtained a **shared-mode lock** (denoted by S) on item *Q*, then *Ti* can read, but cannot write, *Q*.
2. **Exclusive**. If a transaction *Ti* has obtained an **exclusive-mode lock** (denoted by X) on item *Q*, then *Ti* can both read and write *Q*.

	S	**X**
S	true	false
X	false	false

Example:

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

The difference between shared lock and exclusive lock

Shared Lock	**Exclusive Lock**
Shared lock is used for when the transaction wants to perform read operation.	Exclusive lock is used when the transaction wants to perform both read and write operation.
Any number of transactions can hold shared lock on an item.	But exclusive lock can be hold by only one transaction.
Using shared lock data item can be viewed.	Using exclusive lock data can be inserted or deleted.

2. Two-Phase Locking (2PL) Protocol

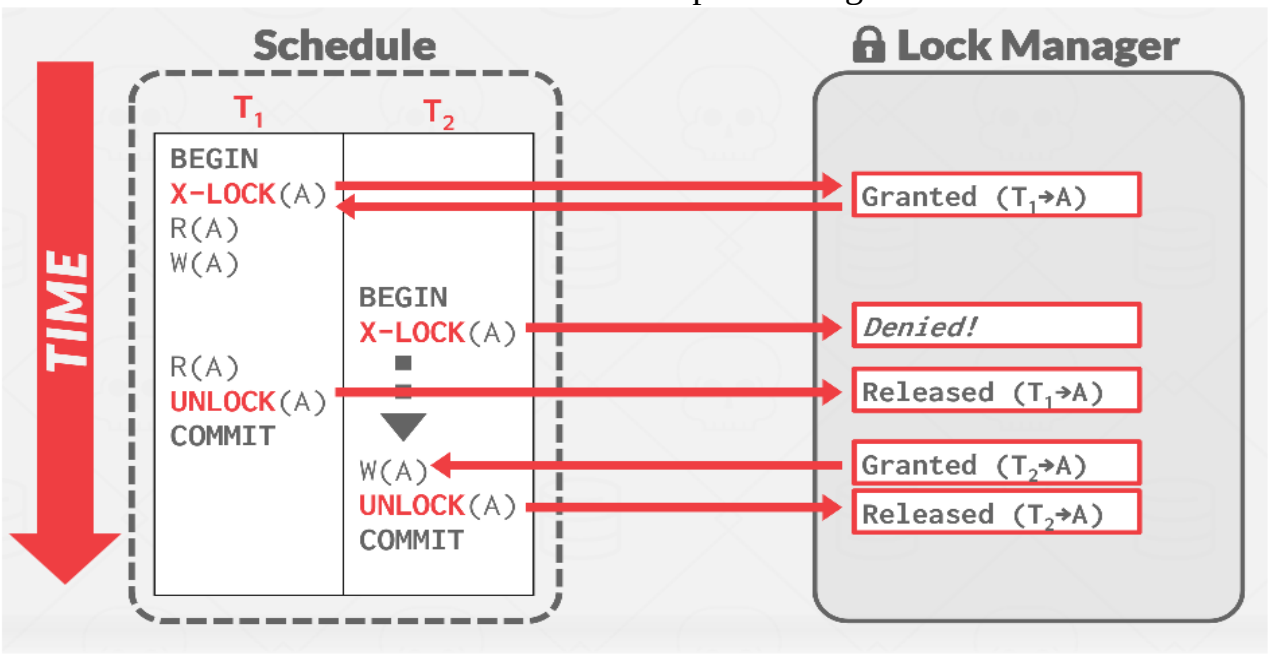
One protocol that ensures serializability is the **Two-Phase Locking Protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. Growing Phase. A transaction may obtain locks, but may not release any lock.

2. Shrinking Phase. A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

Let's see a transaction implementing 2-PL.



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

Lock Point

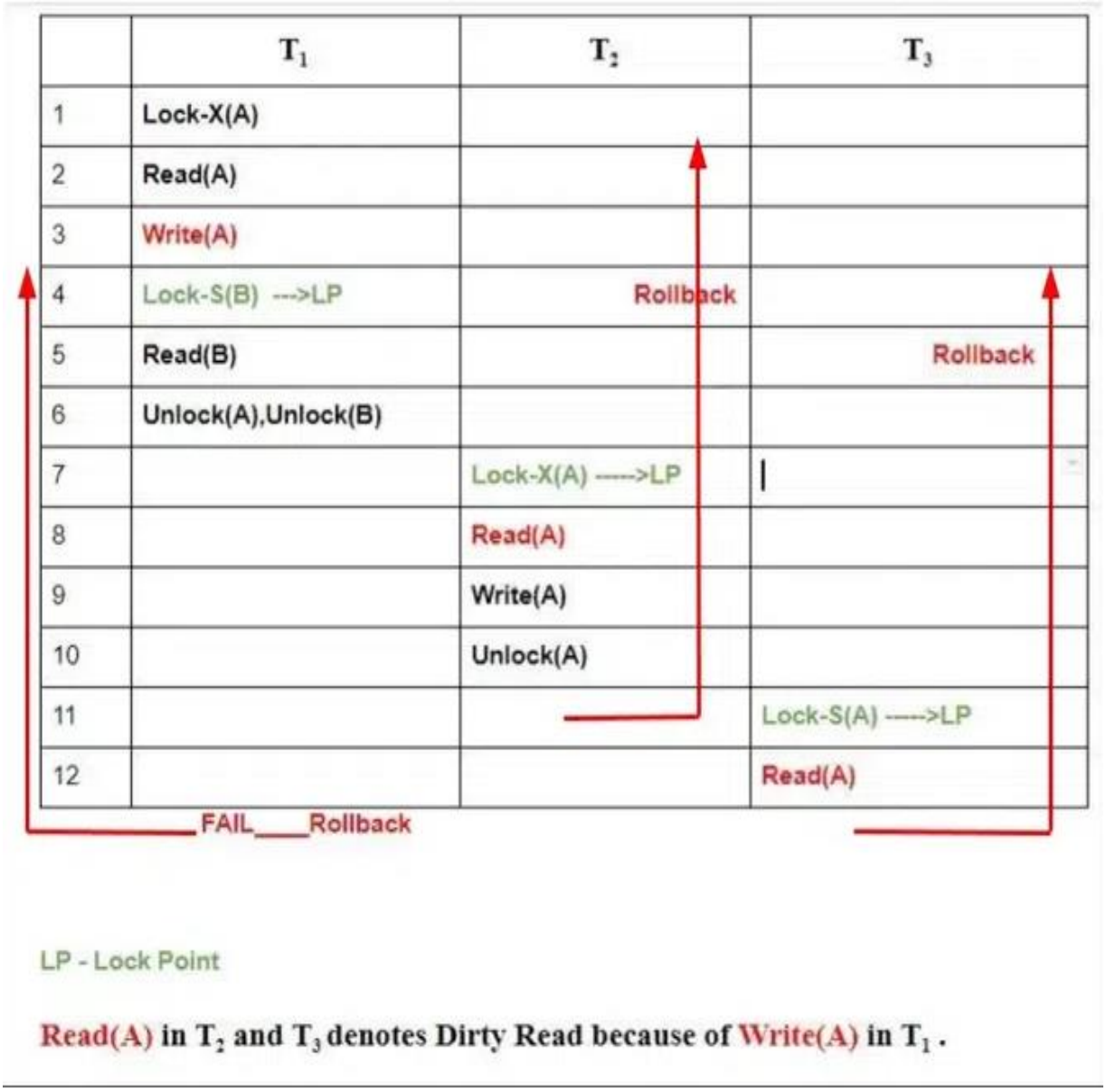
The Point at which the growing phase ends, i.e., when a transaction takes the final lock it needs to carry on its work. Now look at the schedule, you'll surely understand. I have said that 2-PL ensures serializability, but there are still some drawbacks of 2-PL.

Let's glance at the drawbacks.

- Cascading Rollback is possible under 2-PL.
- Deadlocks and Starvation are possible.

Cascading Rollbacks in 2-PL

Let's see the following Schedule



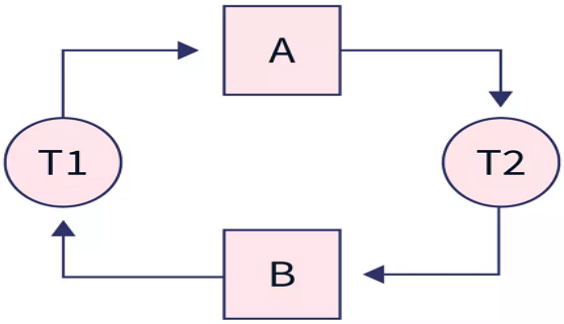
Take a moment to analyze the schedule. Because of Dirty Read in T2 and T3 in lines 8 and 12 respectively, when T1 failed we have to roll back others also. Hence, Cascading Rollbacks are possible in 2-PL.

Deadlock in 2-PL

Consider this simple example; it will be easy to understand. Say we have two transactions T1 and T2.

Schedule: Lock-X1(A) Lock-X2(B) Lock-X1(B) Lock-X2(A)

Drawing the precedence graph, you may detect the loop. So Deadlock is also possible in 2-PL.



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 may be because of the protocols and other restrictions we may put on the schedule to ensure serializability, deadlock freedom, and other factors. The above-mentioned type of 2-PL is called Basic 2PL. To sum it up it ensures Conflict Serializability but does not prevent Cascading Rollback and Deadlock.

Problem with Two-Phase Locking

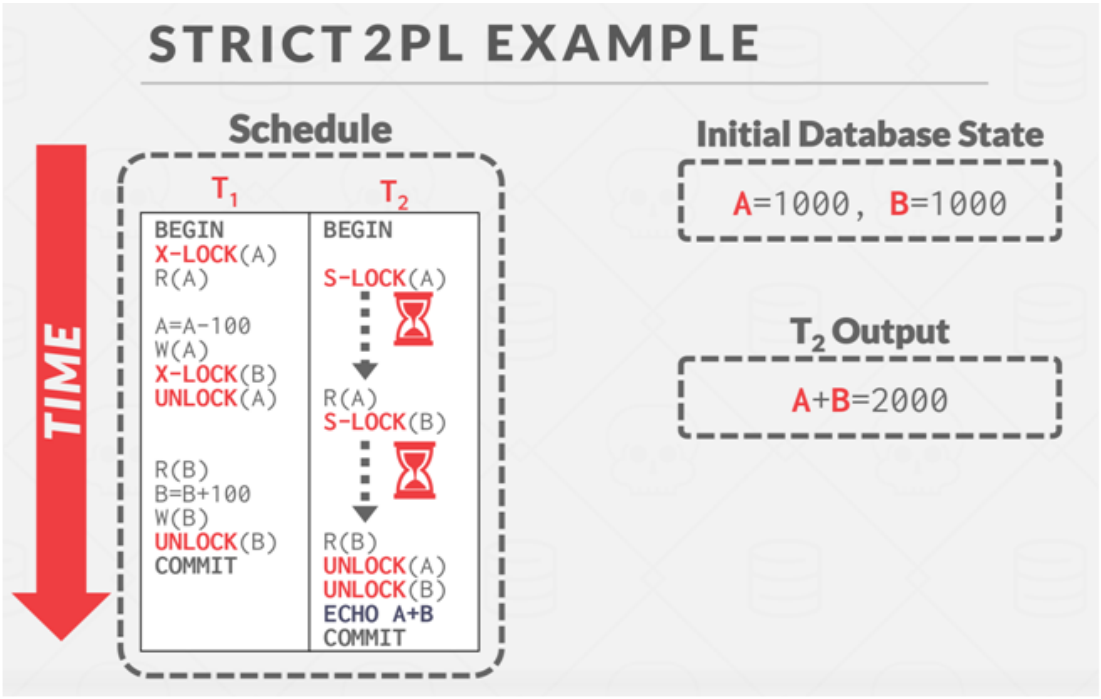
- It does not insure recoverability which can be solved by strict two-phase locking and rigorous two-phase locking.
- It does not ensure a cascade-less schedule which can be solved by strict two-phase locking and rigorous two-phase locking.

Two-Phase Locking is further classified into two types:

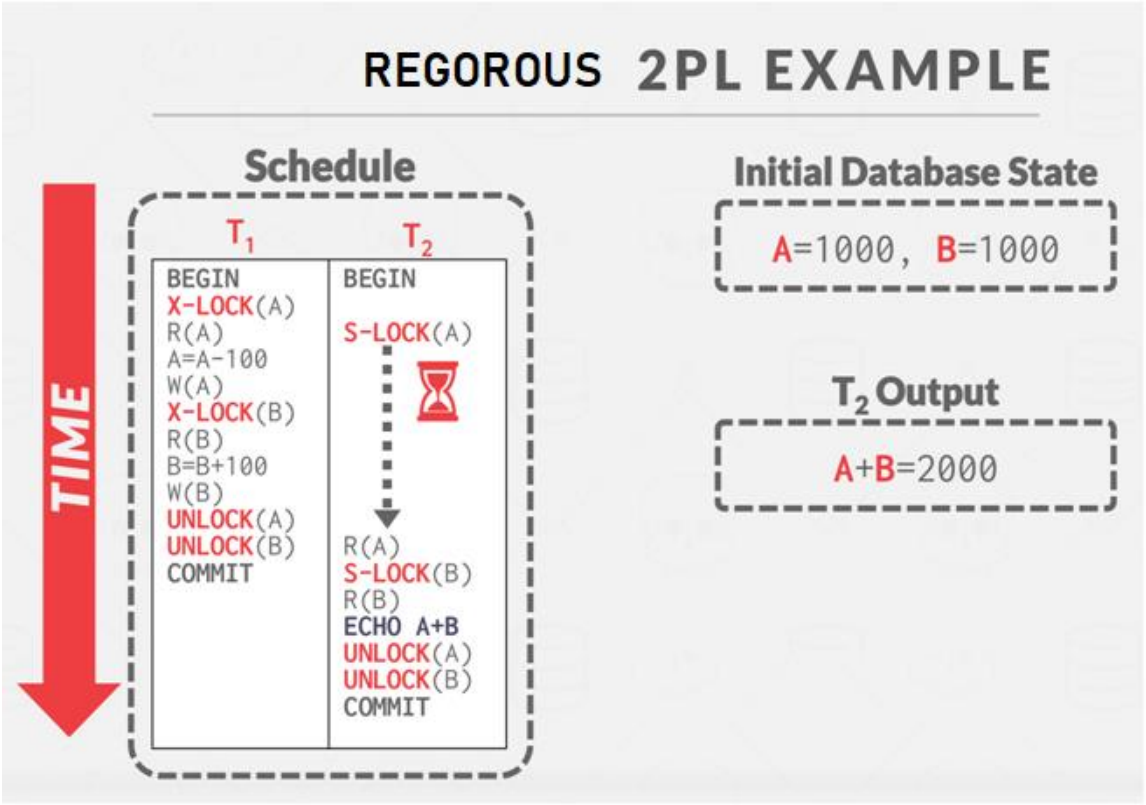
1. Strict Two-Phase Locking Protocol
2. Rigorous Two-Phase Locking Protocol

1. **Strict Two-Phase Locking Protocol** :

- The transaction can release the shared lock after the lock point.
- Each transaction should hold all **Exclusive(X) Locks** until the Transaction is Commited or aborted.
- In strict two-phase locking protocol, if one transaction rollback then the other transaction should also have to roll back. The transactions are dependent on each other. This is called **Cascading schedule**.



2. **Rigorous Two-Phase Locking Protocol** :
   - The transaction cannot release either of the locks, i.e., neither shared lock nor exclusive lock.
   - Serializability is guaranteed in a Rigorous two-phase locking protocol.
   - Deadlock is not guaranteed in the rigorous two-phase locking protocol.



Deadlock Handling

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions $\{T_0, T_1, \ldots, T_n\}$ such that T_0 is waiting for a data item that T_1 holds, and T_1 is waiting for a data item that T_2 holds, and . . . , and T_{n-1} is waiting for a data item that T_n holds, and T_n is waiting for a data item that T_0 holds. None of the transactions can make progress in such a situation.

There are **two principal methods** for dealing with the deadlock problem.

1. We can use a **Deadlock Prevention** protocol to ensure that the system will *never* enter a deadlock state.
2. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **Deadlock Detection** and **Deadlock Recovery** scheme.

1. Deadlock Prevention

It ensures that the system will *never* enter a deadlock state.
Two different deadlock-prevention schemes using timestamps have been proposed.

1. Wait-Die Scheme
2. Wound–Wait Scheme

1. **Wait-Die Scheme:**

In the **Wait-Die** scheme, transactions are assigned **timestamps** when they start, and these timestamps are used to decide the behavior of transactions when they attempt to acquire resources that are already held by other transactions. The core idea is to **allow younger transactions to wait** for resources but **force older transactions to be aborted (killed)** if they request a resource held by a younger transaction.

- **Wait**: When a **younger transaction** requests a resource that is currently held by an **older transaction**, the younger transaction is allowed to **wait** for the resource to become available.
- **Die**: When an **older transaction** requests a resource that is currently held by a **younger transaction**, the older transaction is **killed** (aborted), since it is less preferred in terms of priority.

Summary of Steps:

Time	T1	T2	Action
t1	Requests R1		T1 acquires R1.
t2		Requests R2	T2 acquires R2.
t3	Requests R2		T1 waits (because T2 holds R2).
t4		Requests R1	T2 is aborted (because T1 holds R1). T2 rolls back.
t5			T1 proceeds (after T2 is aborted).

2. Wound–Wait Scheme:

It works by assigning a **priority** to transactions based on their **timestamps** (i.e., the order in which they start). The transactions with **earlier start times** are given higher priority (older transactions), and those with **later start times** are given lower priority (younger transactions).

The Wound-Wait scheme uses the **timestamp** of a transaction to decide its behavior when it requests a resource that is currently held by another transaction:

1. **Wound**: If a **younger transaction** requests a resource held by an **older transaction**, the younger transaction is **aborted** (wounded). It will be restarted later and will try again to acquire the resource.
2. **Wait**: If an **older transaction** requests a resource held by a **younger transaction**, the older transaction is allowed to **wait** for the resource to be released by the younger transaction.

Summary of Steps:

Time	Transaction	Action	Reason	Result
t1	T1	Requests R2	T1 is older, so it waits for T2 to release R2.	T1 is waiting for T2 to release R2.
t2	T2	Requests R1	T2 is younger and requests resource held by T1.	T2 is aborted (wounded), rolled back.
t3	T2	Restarts and tries to request R1 again.	After aborting, T2 restarts and tries to acquire R1.	T2 waits for T1 to release R1.

2. Deadlock Detection and Recovery

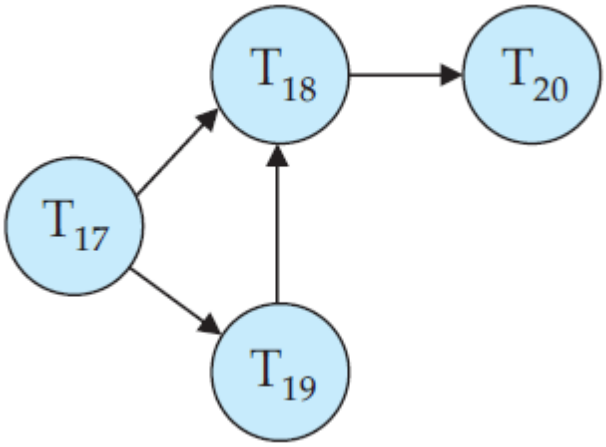
Deadlock Detection allows deadlocks to occur but provides mechanisms to **detect** them and then take action to **recover** from them.

1. **Deadlock Detection**: Involves monitoring the system to check for the presence of deadlocks, usually by analyzing resource allocation and transaction states.
2. **Deadlock Recovery**: Once a deadlock is detected, recovery techniques are employed to break the deadlock by aborting or rolling back one or more transactions involved in the deadlock cycle.

1. Deadlock Detection

Deadlock detection in a DBMS typically involves using the **Wait-for Graph** (WFG) to model the relationships between transactions and the resources they are waiting for.

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 in the system.

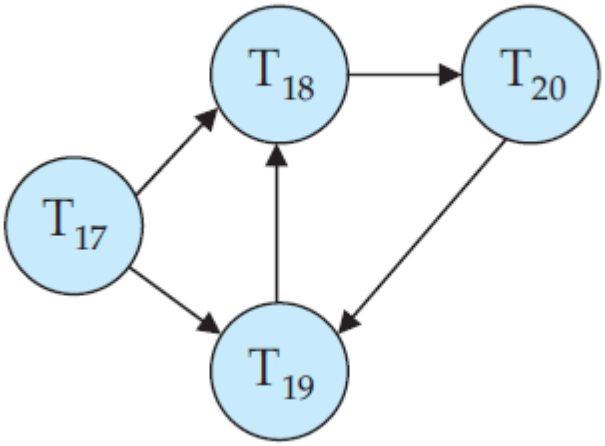


Wait-for graph with no cycle.

Each element in the set E of edges is an ordered pair $Ti \rightarrow Tj$. If $Ti \rightarrow Tj$ is in E, then there is a directed edge from transaction Ti to Tj , implying that transaction Ti is waiting for transaction Tj to release a data item that it needs.

When transaction Ti requests a data item currently being held by transaction Tj , then the edge $Ti \rightarrow Tj$ is inserted in the wait-for graph. This edge is removed only when transaction Tj is no longer holding a data item needed by transaction Ti .

A **deadlock** exists when a cycle is detected in the wait-for graph, meaning that there is a circular chain of transactions waiting for each other's resources.



Wait-for graph with a cycle.

2. Deadlock Recovery

When a detection algorithm determines that a deadlock exists, the system must **recover** from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock.

Three actions need to be taken:

1. Selection of a Victim.

Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock.

We should roll back those transactions that will incur the minimum cost.

a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
b. How many data items the transaction has used.
c. How many more data items the transaction needs for it to complete.
d. How many transactions will be involved in the rollback.

2. **Rollback**.

Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.

There are several strategies for choosing which transaction to abort:

1. Abort one transaction at random
2. Abort the youngest transaction
3. Abort the transaction that has held the least resources
4. Abort the transaction with the lowest priority

3. Starvation

In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation**. We must ensure that a transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Timestamp-Based Protocols

Timestamp-based concurrency control is a method used in database systems to ensure that transactions are executed safely and consistently without conflicts, even when multiple transactions are being processed simultaneously. This approach relies on timestamps to manage and coordinate the execution order of transactions.

With each transaction *Ti* in the system, we associate a unique fixed timestamp, denoted by TS(*Ti*). This timestamp is assigned by the database system before the transaction *Ti* starts execution. If a transaction *Ti* has been assigned timestamp TS(*Ti*), and a new transaction *Tj* enters the system, then TS(*Ti*) < TS(*Tj*).

There are **three types** of Timestamp-Based Protocols

1. Timestamp-Ordering Scheme.
2. The Timestamp-Ordering Protocol
3. Thomas' Write Rule

1. Timestamp-Ordering Scheme.

The most common method for doing so is to use a *timestamp-ordering* scheme. To implement this scheme, we associate with each data item *Q* two timestamp values:

- **W-timestamp**(*Q*) denotes the largest timestamp of any transaction that executed write(*Q*) successfully.
- **R-timestamp**(*Q*) denotes the largest timestamp of any transaction that executed read(*Q*) successfully.

These timestamps are updated whenever a new read(*Q*) or write(*Q*) instruction is executed.

2. The Timestamp-Ordering Protocol

The **timestamp-ordering protocol** ensures that any conflicting read and write operations are executed in timestamp order.

This protocol operates as follows:

1. Suppose that transaction *Ti* issues read(*Q*).

a) If TS(*Ti*) < W-timestamp(*Q*), then *Ti* needs to read a value of *Q* that was already overwritten. Hence, the read operation is rejected, and *Ti* is rolled back.
b) If TS(*Ti*) ≥ W-timestamp(*Q*), then the read operation is executed, and R-timestamp(*Q*) is set to the maximum of R-timestamp(*Q*) and TS(*Ti*).

2. Suppose that transaction *Ti* issues write(*Q*).

a) If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls *Ti* back.
b) If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*. Hence, the system rejects this write operation and rolls *Ti* back.
c) Otherwise, the system executes the write operation and sets W-timestamp(*Q*) to TS(*Ti*).

3. **Thomas' Write Rule**

We now present a modification to the timestamp-ordering protocol that allows greater potential concurrency than does the protocol of **Timestamp-Ordering.**

The modification to the timestamp-ordering protocol, called **Thomas' write rule**, is:

Suppose that transaction *Ti* issues write(*Q*).

a) If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls *Ti* back.
b) If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*. Hence, this write operation can be ignored.
c) Otherwise, the system executes the write operation and setsW-timestamp(*Q*) to TS(*Ti*).

Validation-Based Protocols

The **Validation Protocol** requires that each transaction *Ti* executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction.

The phases are, in order:

1. **Read Phase**: During this phase, the system executes transaction *Ti*. It reads the values of the various data items and stores them in variables local to *Ti*. It performs all write operations on temporary local variables, without updates of the actual database.
2. **Validation Phase**: The validation test (described below) is applied to transaction *Ti* . This determines whether *Ti* is allowed to proceed to the write phase without causing a violation of serializability. If a transaction fails the validation test, the system aborts the transaction.
3. **Write Phase**: If the validation test succeeds for transaction *Ti*, the temporary local variables that hold the results of any write operations performed by *Ti* are copied to the database. Read-only transactions omit this phase.

Each transaction must go through the phases in the order shown. However, phases of concurrently executing transactions can be interleaved.

To perform the validation test, we need to know when the various phases of transactions took place. We shall, therefore, associate three different timestamps with each transaction *Ti* :

1. **Start**(*Ti*), the time when *Ti* started its execution.
2. **Validation**(*Ti*), the time when *Ti* finished its read phase and started its validation phase.
3. **Finish**(*Ti*), the time when *Ti* finished its write phase.

We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp Validation(*Ti*). Thus, the value TS(*Ti*) = Validation(*Ti*) and, if TS(*Tj*) < TS(*Tk*), then any produced schedule must be equivalent to a serial schedule in which transaction *Tj* appears before transaction *Tk*. The reason we have chosen Validation(*Ti*), rather than Start(*Ti*), as the timestamp of transaction *Ti* is that we can expect faster response time provided that conflict rates among transactions are indeed low.

The **validation test** for transaction *Ti* requires that, for all transactions *Tk* with TS(*Tk*) < TS(*Ti*), one of the following two conditions must hold:

1. Finish(*Tk*) < Start(*Ti*). Since *Tk* completes its execution before *Ti* started, the serializability order is indeed maintained.
2. The set of data items written by *Tk* does not intersect with the set of data items read by *Ti*, and *Tk* completes its write phase before *Ti* starts its validation phase (Start(*Ti*) < Finish(*Tk*) < Validation(*Ti*)). This condition ensures that the writes of *Tk* and *Ti* do not overlap. Since the writes of *Tk* do not affect the read of *Ti* , and since *Ti* cannot affect the read of *Tk* , the serializability order is indeed maintained.

As an illustration, consider again transactions *T*25 and *T*26. Suppose that TS(*T*25) < TS(*T*26). Then, the validation phase succeeds in the below schedule.

Note that the writes to the actual variables are performed only after the validation phase of *T*26. Thus, *T*25 reads the old values of *B* and *A*, and this schedule is serializable.

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
< validate>	
display($A + B$)	
	< validate>
	write(B)
	write(A)

A schedule produced by using validation

This validation scheme is called the **Optimistic Concurrency-Control** scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end.

In contrast, **Locking and Timestamp Ordering** are **Pessimistic Concurrency-Control** in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serializable