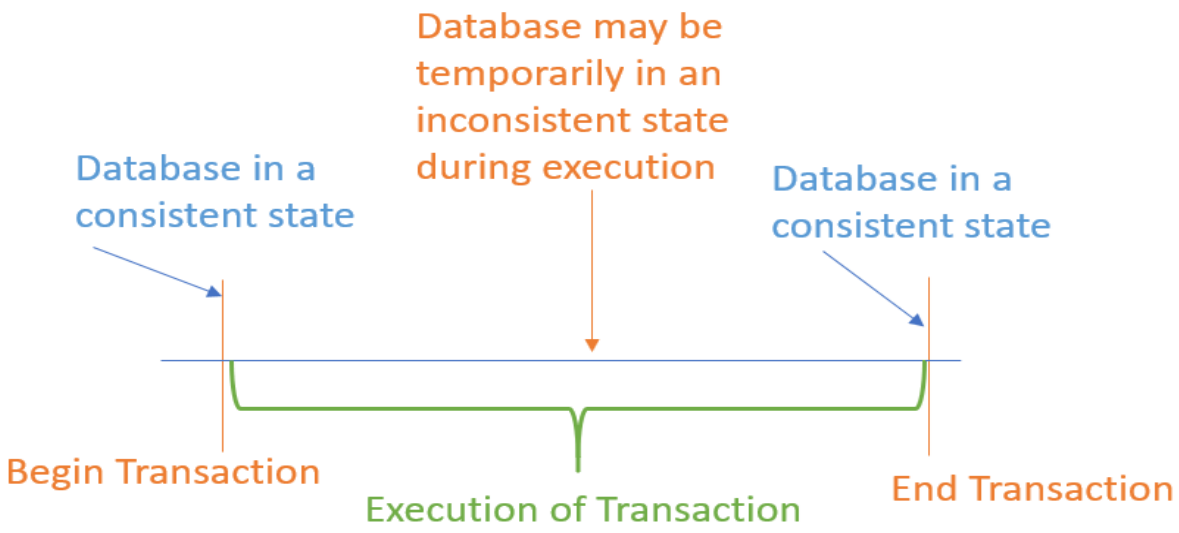
UNIT – IV

Transaction Concept, Transaction State, Implementation of Atomicity and Durability, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for serializability, Lock Based Protocols, Timestamp Based Protocols, Validation- Based Protocols, Multiple Granularity, Recovery and Atomicity, Log–Based Recovery, Recovery with Concurrent Transactions.

TRANSACTION CONCEPT

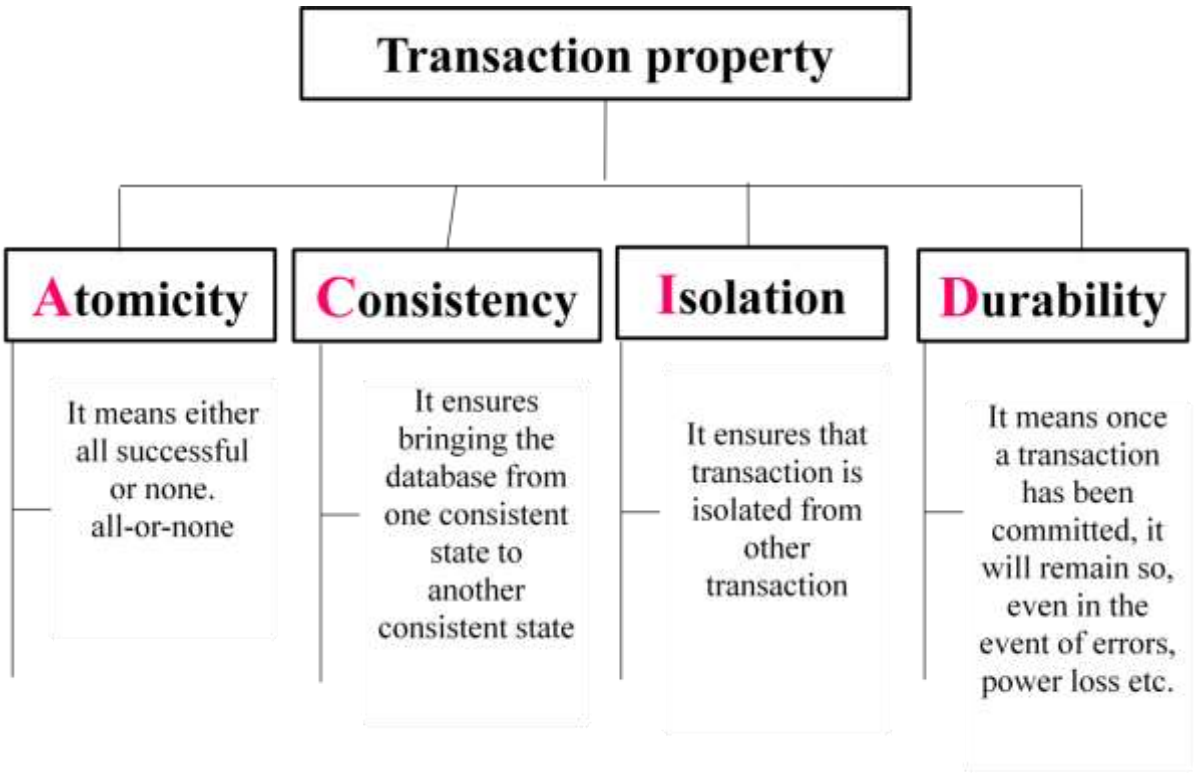
What is Transaction?

- A **transaction** is a **unit** of program execution that accesses and possibly updates various data items.
- A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.
- The transaction consists of all operations executed between the **begin transaction** and **end transaction**.



TRANSACTION PROPERTY/ ACID PROPERTY

- The transaction has the four properties.
    - **Atomicity**
    - **Consistency**
    - **Isolation**
    - **Durability**
- These are used to maintain consistency in a database, before and after the transaction.



A Simple Transaction Model

- Transactions access data using two operations:
  - **read(*X*),** which transfers the data item *X* from the database to a variable, also called *X*, in a buffer in main memory belonging to the transaction that executed the read operation.
  - **write(*X*),** which transfers the value in the variable *X* in the main-memory buffer of the transaction that executed the write to the data item *X* in the database.

Example

- Let *Ti* be a transaction that transfers $50 from account *A* to account *B*.
- This transaction can be defined as:

  ***Ti*: read(*A*);**
  ***A* := *A* - 50;**
  write(*A*);
  read(*B*);
  ***B* := *B* + 50;**
  write(*B*);

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.

Operations of Atomicity

- Atomicity involves the following two operations:
  - **Abort:** If a transaction aborts then all the changes made are not visible.
  - **Commit:** If a transaction commits then all the changes made are visible.

Example

- Suppose that, just before the execution of transaction *Ti*, the values of accounts *A* and *B* are $1000 and $2000, respectively.
- Now suppose that, during the execution of transaction *Ti*, a failure occurs that prevents *Ti* from completing its execution successfully.
- Further, suppose that the failure happened after the write(*A*) operation but before the write(*B*) operation.
- In this case, the values of accounts *A* and *B* reflected in the database are $950 and $2000. The system destroyed $50 as a result of this failure. In particular, we note that the sum *A* + *B* is no longer preserved.
- Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an **inconsistent state**.

A = $1000 *B* =$2000

T1:

```
read(A);
A := A - 50;
write(A);  // Transaction Failed
```

T2:

```
read(B);
B := B + 50;
write(B).
```

Now, *A* = $950 *B* =$2000 (This is inconsistent state)

Solution to Inconsistency Problem

- If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.
- The basic idea behind ensuring atomicity is this:
- The database system keeps track (on disk) of the old values of any data on which a transaction performs a write.
- This information is written to a file called the log.
- If the transaction does not complete its execution, the database system restores the old values from the log to make it appear as though the transaction never executed.

- Ensuring atomicity is the responsibility of the database system; specifically, it is handled by a component of the database called the **recovery system.**

CONSISTENCY

- Consistency means that integrity constraints must be maintained so that the database is consistent before and after the transaction.
- It refers to the correctness of a database.

Example

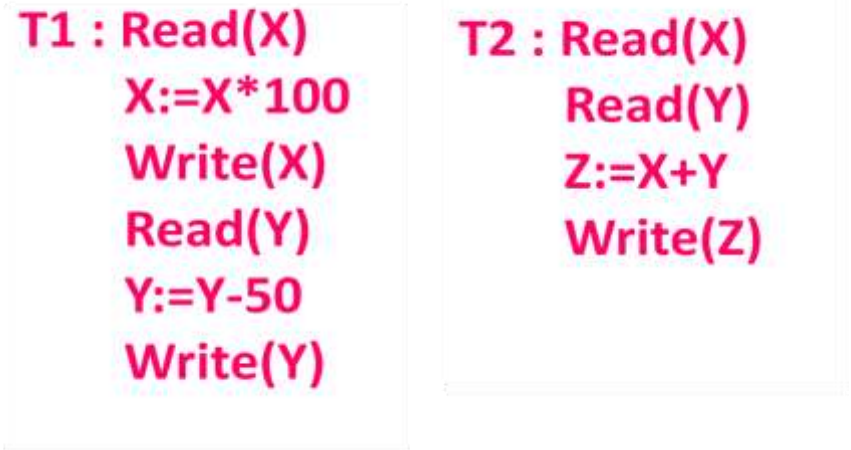
- Referring to the example above,
- The total amount before and after the transaction must be maintained.
  Total **before T1** *A* = $1000 *B* =$2000
  Total **after T2** *A* = $950 *B* =$2050
- Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result T is incomplete.

ISOLATION

- Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.
- Isolation ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state.

Example

- Let **X**= 500, **Y** = 500. Consider two transactions **T1** and **T2.**



- Suppose **T1** has been executed till **Read (Y)** and then **T2** starts.
- As a result , interleaving of operations takes place due to which **T2** reads correct value of **X** but incorrect value of **Y** and sum computed by

  T2: (X+Y = 50, 000+500=50, 500 is thus not consistent with the sum at end of transaction:
  T1: (X+Y = 50, 000 + 450 = 50, 450)

This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

Durability

- Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure can result in a loss of data corresponding to this transfer of funds.
- The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.
- Assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost.
- The durability is guaranteed by ensuring that either:
  1. **The updates carried out by the transaction have been written to disk before the transaction completes.**
  2. **Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.**

TRANSACTION STATE

What is Transaction State?

- A transaction goes through many different states throughout its life cycle.
- These states are called as transaction states.

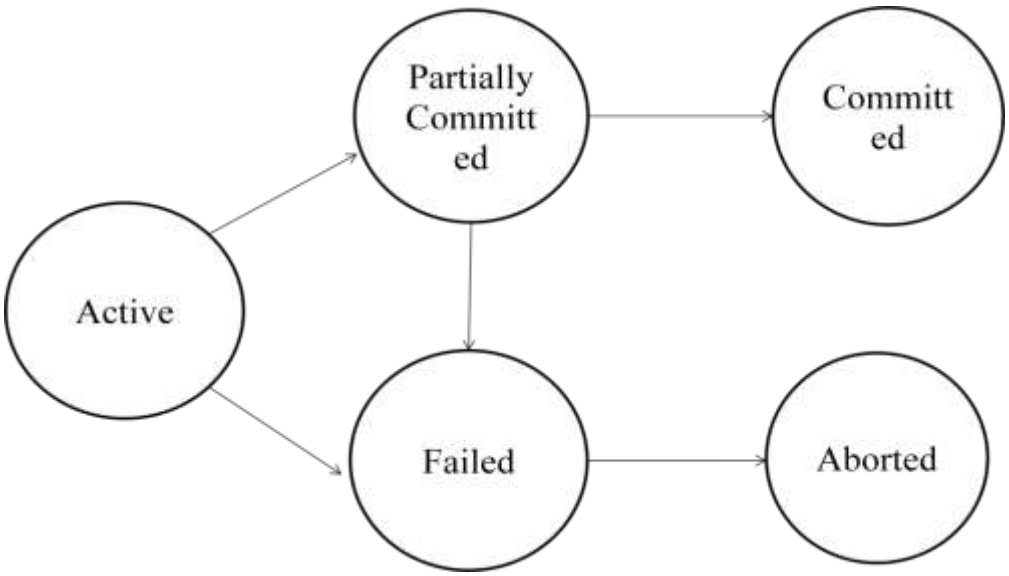
Transaction States

- A transaction must be in one of the following states:
  - **Active**, the initial state; the transaction stays in this state while it is executing.
  - **Partially committed**, after the final statement has been executed.
  - **Failed**, after the discovery that normal execution can no longer proceed.
  - **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
  - **Committed**, after successful completion.

State Diagram of a transaction

- A transaction has **committed** only if it has entered the committed state.
- A transaction has **aborted** only if it has entered the aborted state.
- A transaction is said to have **terminated** if it has either committed or aborted.

A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state.



- At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.
- The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure.
- When the last of this information is written out, the transaction enters the committed state.
- A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors).
- Such a transaction must be rolled back. Then, it enters the aborted state.
- At this point, the system has two options:
    - It can **restart** the transaction, but only if the transaction was aborted as a result of some **hardware or software error** that was not created through the internal logic of the transaction.
    - A restarted transaction is considered to be a new transaction.

- It can **kill** the transaction. It usually does so because of some **internal logical error** that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

TRANSACTION ATOMICITY AND DURABILITY

- A transaction may not always complete its execution successfully. Such a transaction is termed **aborted**.
- To ensure the atomicity property, an aborted transaction must have no effect on the state of the database.

- Thus, any changes that the aborted transaction made to the database must be undone.
- Once the changes caused by an aborted transaction have been undone, it means the transaction has been **rolled back**.
- It is part of the responsibility of the recovery scheme to manage transaction aborts. This is done typically by maintaining a **log**.
- Each database modification made by a transaction is first recorded in the log.
- The identifier of the transaction performing the modification is recorded, the identifier of the data item being modified, and both the old value (prior to modification) and the new value (after modification) of the data item.
- Only then is the database itself modified.
- Maintaining a log provides the possibility of redoing a modification to ensure atomicity and durability as well as the possibility of undoing a modification to ensure atomicity in case of a failure during transaction execution.
- A transaction that completes its execution successfully is said to be **committed**.
- A committed transaction that has performed updates transforms the database into a new consistent state, which must persist even if there is a system failure.
- Once a transaction has committed, its effects cannot be undone by aborting it.
- The only way to undo the effects of a committed transaction is to execute a **compensating transaction**.
- For instance, if a transaction added $20 to an account, the compensating transaction would subtract $20 from the account. However, it is not always possible to create such a compensating transaction.
- Therefore, the responsibility of writing and executing a compensating transaction is left to the user, and is not handled by the database system.

Observable external writes

- When dealing with **observable external writes**, such as writes to a user‘s screen, or sending email care should be taken.
- Once such a write has occurred, it cannot be erased, since it may have been seen external to the database system.
- Most systems allow such writes to take place only after the transaction has entered the committed state.
- One way to implement such a scheme is for the database system to store any value associated with such external writes temporarily in a special relation in the database, and to perform the actual writes only after the transaction enters the committed state.
- If the system should fail after the transaction has entered the committed state, but before it could complete the external writes, the database system will carry out the external writes (using the data in nonvolatile storage) when the system is restarted.
- Handling external writes can be more complicated in some situations.

Example

- Suppose the external action is that of dispensing cash at an automated teller machine, and the system fails just before the cash is actually dispensed (we assume that cash can be dispensed atomically). It makes no sense to dispense cash when the system is restarted,

since the user may have left the machine. In such a case a compensating transaction, such as depositing the cash back in the user‘s account, needs to be executed when the system is restarted.

- Consider a user making a booking over the Web. It is possible that the database system or the application server crashes just after the booking transaction commits. It is also possible that the network connection to the user is lost just after the booking transaction commits. In either case, even though the transaction has committed, the external write has not taken place. To handle such situations, the application must be designed such that when the user connects to the Web application again, she will be able to see whether her transaction had succeeded or not.
- For certain applications, it may be desirable to allow active transactions to display data to users, particularly for long-duration transactions that run for minutes or hours. Unfortunately, we cannot allow such output of observable data unless we are willing to compromise transaction atomicity.

CONCURRENT EXECUTIONS

What is Concurrent Executions?

- In transaction-processing, a system usually allows multiple transactions to run concurrently.

Advantages of concurrent execution of a transaction

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.
- Moreover, it also reduces the **average response time**: the average time for a transaction to be completed after it has been submitted.

Concurrency Control Schemes

- When several transactions run concurrently, the isolation property may be violated, resulting in database consistency being destroyed despite the correctness of each individual transaction.
- The database system must **control the interaction among the concurrent transactions** to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called **concurrency-control schemes**.

Example

Consider again the simplified banking system of which has several accounts, and a set of transactions that access and update those accounts. Let *T*1 and *T*2 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*. It is defined as:

***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. 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 T1 is followed by T2

- Similarly, if the transactions are executed one at a time in the order *T*2 followed by *T*1, then the corresponding execution sequence. Again, as expected, the sum *A* + *B* is preserved, and the final values of accounts *A* and *B* are $850 and $2150, respectively.

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

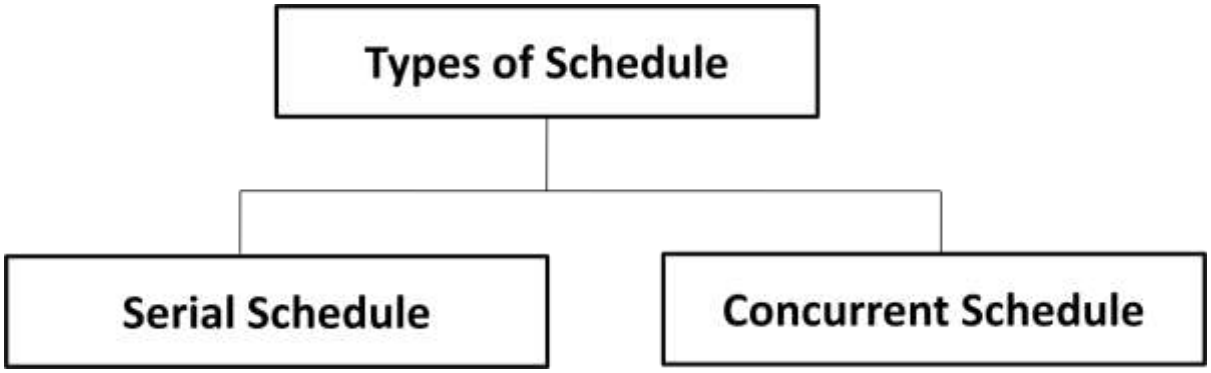
Schedule 2—a serial schedule in which *T*2 is followed by *T*1

Schedule

- The execution sequences just described are called **schedules**.
- A schedule is a **series of operations** from one or more transactions.

- 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.

Types of Schedule



Serial Schedules

- When one transaction completely executes before starting another transaction, the schedule is called serial schedule.
- A serial schedule is always consistent.
- A serial schedule has low throughput and less resource utilization.

Example

- In transaction *T*1, the instruction write(*A*) must appear before the instruction read(*B*), in any valid schedule.
- Schedules the **commit** operation to indicate that the transaction has entered the committed state. For example, in transaction *T*1, the instruction write(*A*) must appear before the instruction read(*B*), in any valid schedule.
- The first execution sequence (*T*1 followed by *T*2) as schedule 1, and to the second execution sequence (*T*2 followed by *T*1) as schedule 2.
- 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.
- For a set of *n* transactions, there exist *n* factorial (*n*!) different valid serial schedules.

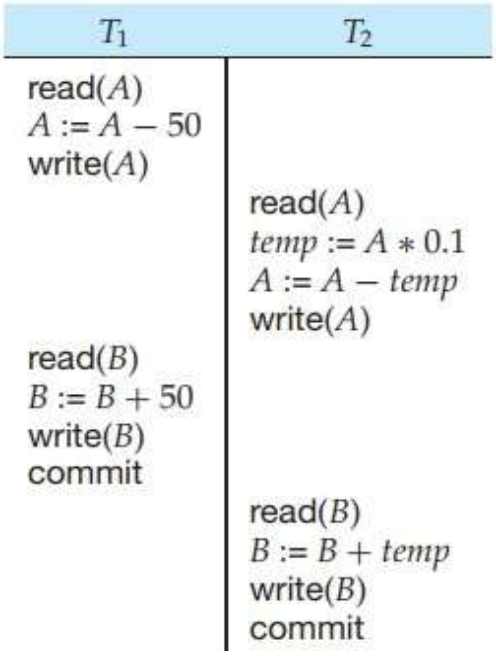
Concurrent Execution

- When operations of a transaction are **interleaved with operations of other transactions** of a schedule, the schedule is called Concurrent schedule.

- When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial.
- If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on.
- With multiple transactions, the CPU time is shared among all the transactions.

Example

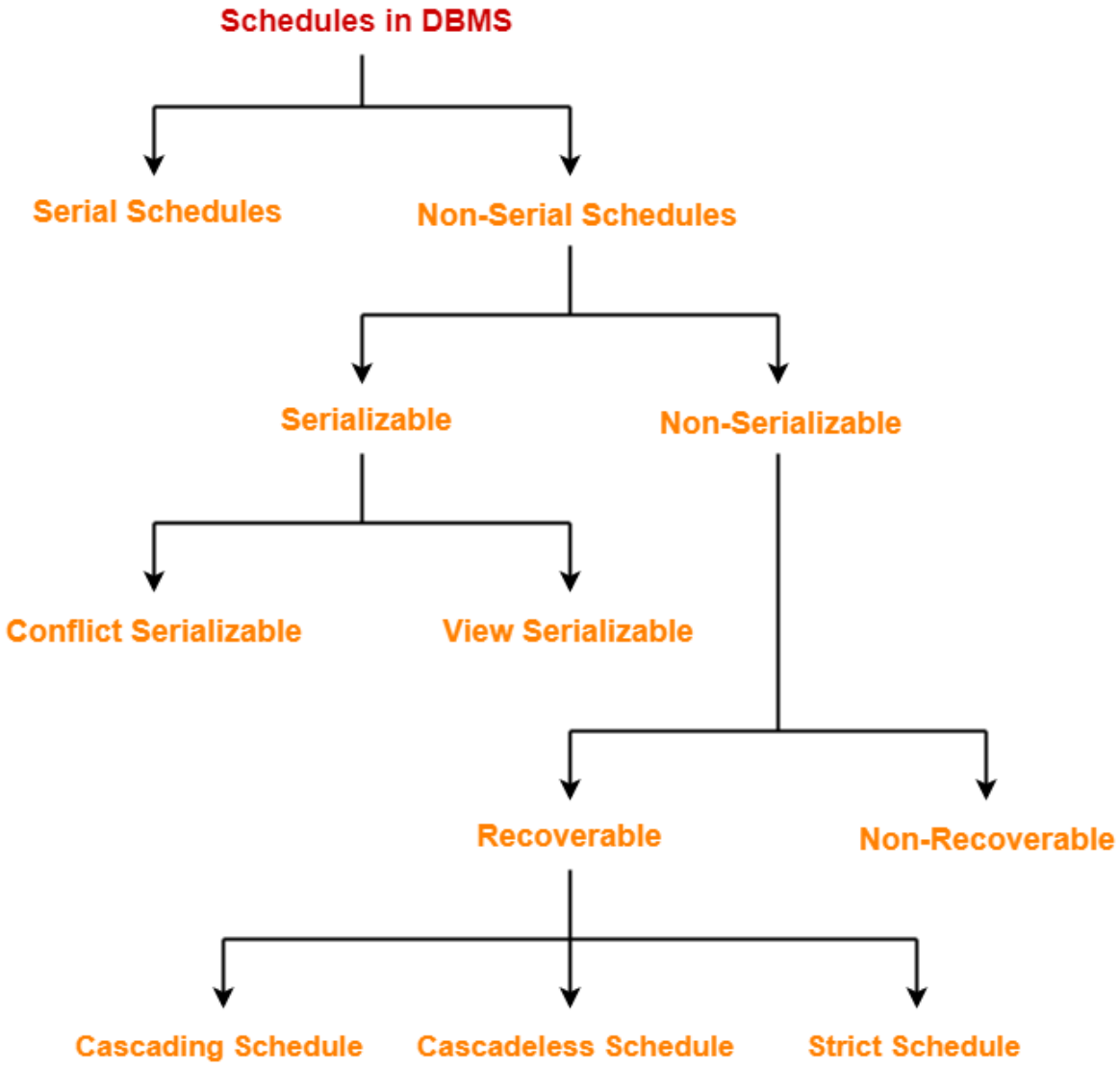
- Suppose that the two transactions are executed concurrently.
- After this execution takes place, we arrive at the same state as the one in which the transactions are executed serially in the order $T1$ followed by $T2$.
- The sum $A + B$ is indeed preserved.



Schedule 3—a concurrent schedule equivalent to schedule 1

- If the system should fail after the transaction has entered the committed state, but before it could complete the external writes, the database system will carry out the external writes (using the data in nonvolatile storage) when the system is restarted.
- Handling external writes can be more complicated in some situations.

SERIALIZABILITY



What is Serializability?

- Non-serial schedules do not wait for one transaction to complete for the other one to begin.
- Some non-serial schedules may lead to inconsistency of the database.
- Serializability is a concept that helps to **identify which non-serial schedules are correct** and will maintain the consistency of the database.

What is serializable schedule?

- If a given non-serial schedule of _n‘ transactions is equivalent to some serial schedule of _n‘ transactions, then it is called as a **serializable schedule**.
- If a schedule of concurrent _n‘ transactions can be converted into an equivalent serial schedule. Then we can say that the schedule is serializable. And this property is known as serializability.

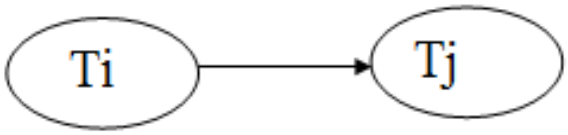
Serial Schedules Vs Serializable Schedules

Serial Schedules	Serializable Schedules
No concurrency is allowed. Thus, all the transactions necessarily execute serially one after the other.	Concurrency is allowed. Thus, multiple transactions can execute concurrently.
Serial schedules lead to less resource utilization and CPU throughput.	Serializable schedules improve both resource utilization and CPU throughput.
Serial Schedules are less efficient as compared to serializable schedules.	Serializable Schedules are always better than serial schedules.

Testing of Serializability

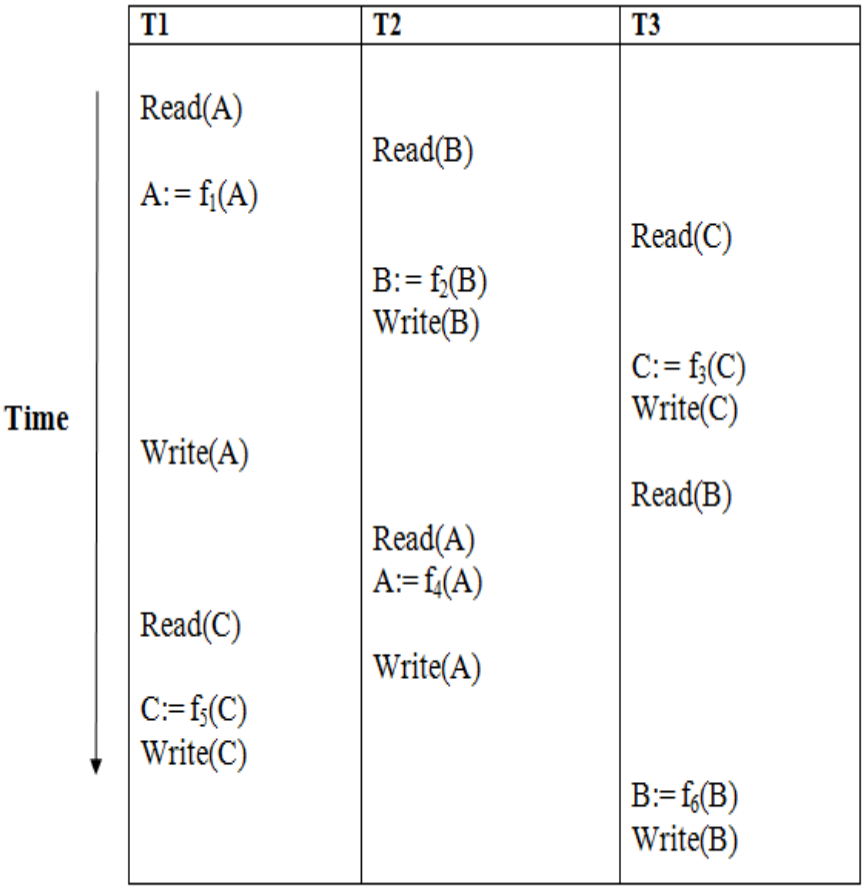
- To test the serializability of a schedule, the serialization graph is used.
- Assume a schedule S. For S, construct a graph known as **precedence graph**.
- This graph has a pair G = (V, E), where V consists a set of vertices, and E consists a set of edges.
- The set of vertices is used to contain all the transactions participating in the schedule.
- The set of edges is used to contain all edges Ti-> Tj for which one of the three conditions holds:
    - **Create a node Ti → Tj, if Ti executes write (Q) before Tj executes read (Q).**
    - **Create a node Ti → Tj, if Ti executes read (Q) before Tj executes write (Q).**
    - **Create a node Ti → Tj, if Ti executes write (Q) before Tj executes write (Q).**

Precedence graph for schedule s



- If a precedence graph contains a single edge Ti → Tj, then all the instructions of Ti are executed before the first instruction of Tj is executed.
- If a precedence graph for schedule S **contains a cycle**, then S is **non-serializable**.
- If the precedence graph has **no cycle**, then S is known as **serializable.**

Example



Schedule S1

Read(A): In T1, no subsequent writes to A, so no new edges

Read(B): In T2, no subsequent writes to B, so no new edges

Read(C): In T3, no subsequent writes to C, so no new edges

Write(B): B is subsequently read by T3, so add edge T2 → T3

Write(C): C is subsequently read by T1, so add edge T3 → T1

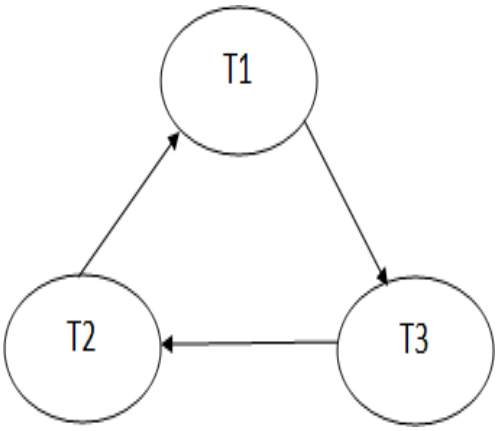
Write(A): A is subsequently read by T2, so add edge T1 → T2

Write(A): In T2, no subsequent reads to A, so no new edges

Write(C): In T1, no subsequent reads to C, so no new edges

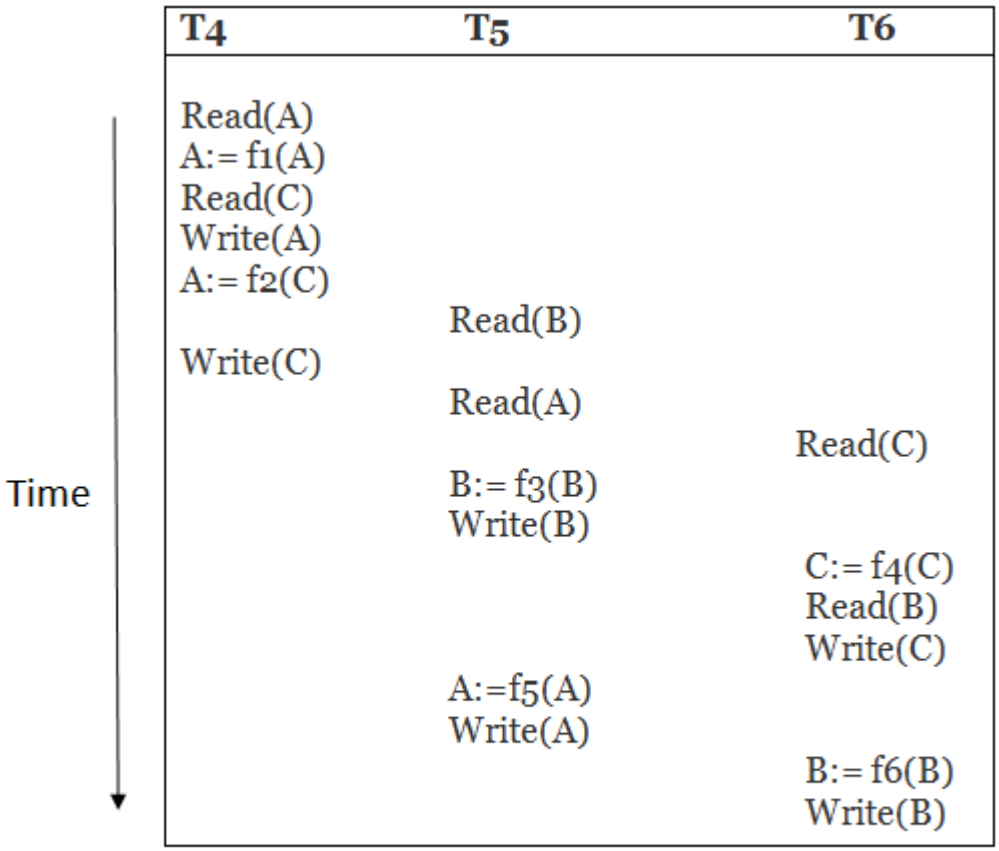
Write(B): In T3, no subsequent reads to B, so no new edges

Precedence graph for schedule S1



The precedence graph for schedule S1 contains a cycle that's why Schedule S1 is non-serializable.

Example 2

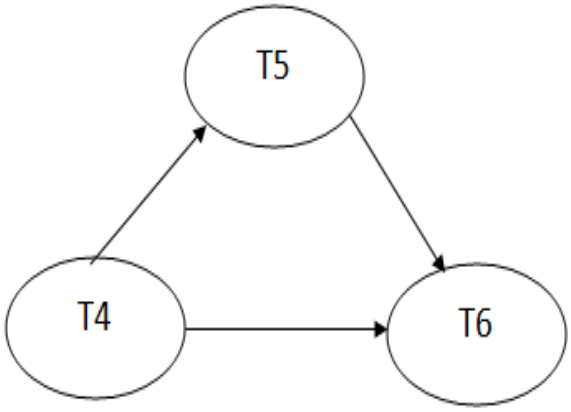


Schedule S2

- **Read(A):** In T4, no subsequent writes to A, so no new edges
- **Read(C):** In T4, no subsequent writes to C, so no new edges
- **Write(A):** A is subsequently read by T5, so add edge T4 → T5
- **Read(B):** In T5,no subsequent writes to B, so no new edges

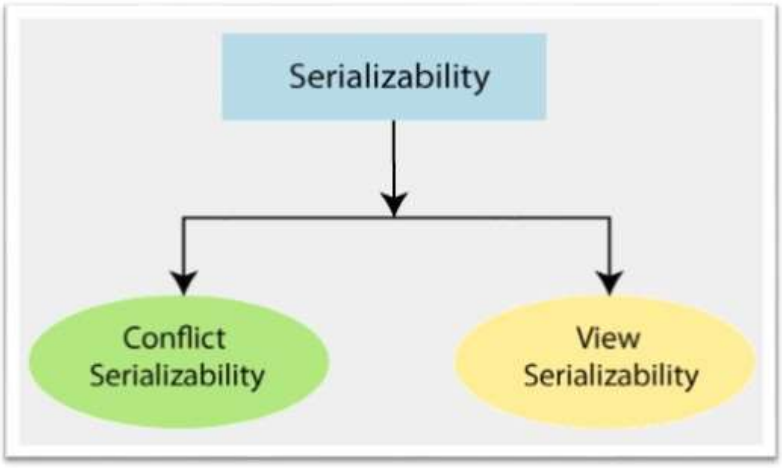
- **Write(C):** C is subsequently read by T6, so add edge T4 → T6
- **Write(B):** A is subsequently read by T6, so add edge T5 → T6
- **Write(C):** In T6, no subsequent reads to C, so no new edges
- **Write(A):** In T5, no subsequent reads to A, so no new edges
- **Write(B):** In T6, no subsequent reads to B, so no new edges

Precedence graph for schedule S2



The precedence graph for schedule S2 contains no cycle that's why ScheduleS2 is serializable.

Types of Serializability



Conflict Serializability

- If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then it is called as a **conflict serializable schedule**.

Example

Let us consider a schedule *S* in which there are two consecutive instructions, *I* and *J* , of transactions *Ti* and *Tj*, respectively ($I \neq j$). If *I* and *J* refer to different data items, then we

can swap *I* and *J* without affecting the results of any instruction in the schedule. However, if *I* and *J* refer to the same data item *Q*, then the order of the two steps may matter.

Cases

1. *I* = read(*Q*), *J* = read(*Q*). The order of *I* and *J* does not matter, since the same value of *Q* is read by *Ti* and *Tj*, regardless of the order.
2. *I* = read(*Q*), *J* = write(*Q*). If *I* comes before *J* , then *Ti* does not read the value of *Q* that is written by *Tj* in instruction *J* . If *J* comes before *I*, then *Ti* reads the value of *Q* that is written by *Tj*. Thus, the order of *I* and *J* matters.
3. *I* = write(*Q*), *J* = read(*Q*). The order of *I* and *J* matters for reasons similar to those of the previous case.
4. *I* = write(*Q*), *J* = write(*Q*). Since both instructions are write operations, the order of these instructions does not affect either *Ti* or *Tj*. However, the value obtained by the next read(*Q*) instruction of *S* is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(*Q*) instruction after *I* and *J* in *S*, then the order of *I* and *J* directly affects the final value of *Q* in the database state that results from schedule *S*.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3 — showing only the read and write instructions.

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Schedule 5—schedule 3 after swapping of a pair of instructions

Since the write(*A*) instruction of *T*2 in schedule 3 does not conflict with the read(*B*) instruction of *T*1, we can swap these instructions to generate an equivalent schedule, schedule

5. Regardless of the initial system state, schedules 3 and 5 both produce the same final system state.

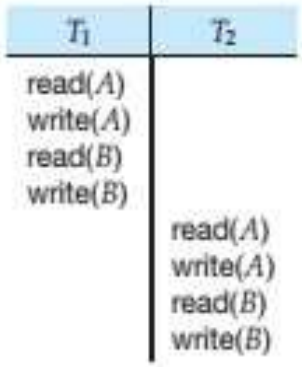
Continue to swap nonconflicting instructions:

• Swap the read(*B*) instruction of *T*1 with the read(*A*) instruction of *T*2.

• Swap the write(*B*) instruction of *T*1 with the write(*A*) instruction of *T*2.

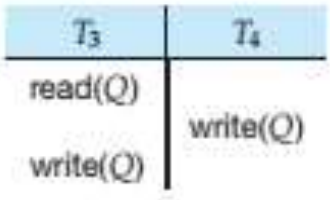
• Swap the write(*B*) instruction of *T*1 with the read(*A*) instruction of *T*2.

If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of non conflicting instructions, we say that *S* and *S'* are **conflict equivalent**. The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule. Thus, schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.



Schedule 6—a serial schedule that is equivalent to schedule 3

Finally, consider schedule 7 it consists of only the significant operations (that is, the read and write) of transactions *T*3 and *T*4. This schedule is not conflict serializable, since it is not equivalent to either the serial schedule <*T*3, *T*4> or the serial schedule <*T*4, *T*3>.



Schedule 7

Precedence graph for (a) schedule 1 and (b) schedule 2

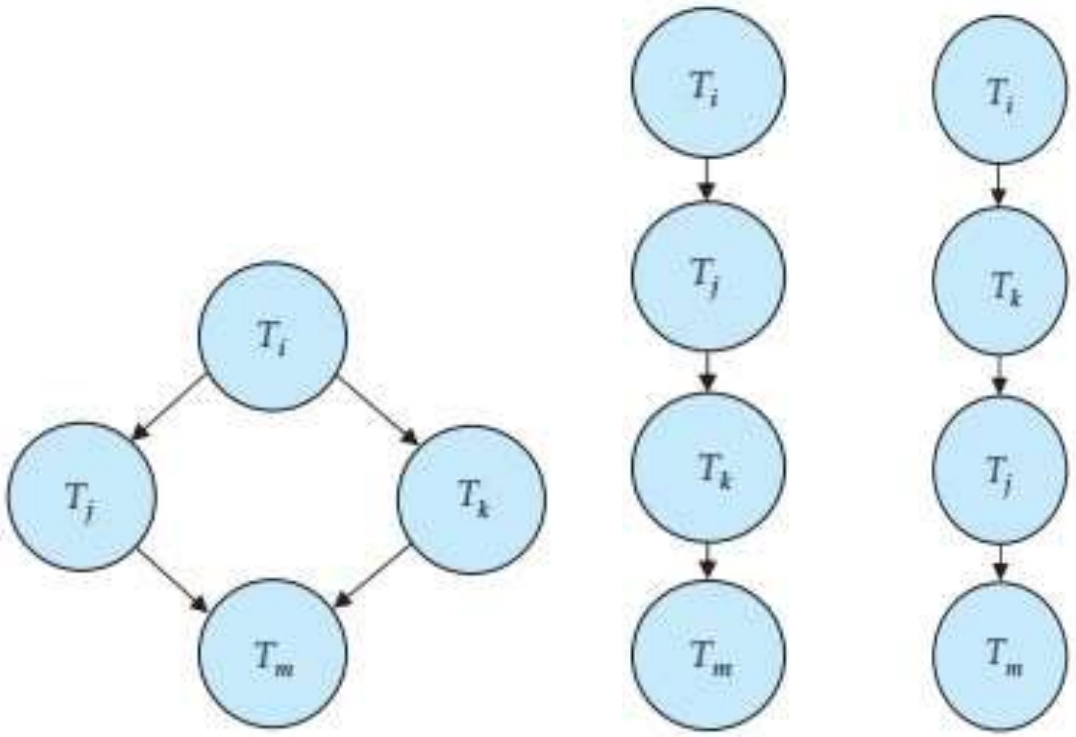
Consider a schedule *S*. Construct a directed graph, called a **precedence graph**, from *S*. This graph consists of a pair *G* = (*V*, *E*), where *V* is a set of vertices and *E* is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges Ti → *Tj* for which one of three conditions holds:

- **Ti executes write(Q) before Tj executes read(Q).**
- **Ti executes read(Q) before Tj executes write(Q).**
- **Ti executes write(Q) before Tj executes write(Q).**

If an edge *Ti* → *Tj* exists in the precedence graph, then, in any serial schedule *S'* equivalent to *S*, *Ti* must appear before *Tj*.

Topological sorting

- A **serializability order** of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called **topological sorting**.
- There are, in general, several possible linear orders that can be obtained through a topological sort.



View Serializability

- A Schedule is called view serializable if it is view equal to a serial schedule (no overlapping transactions).

View Equivalent

- Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:
    - **Initial Read:** An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.
    - Below two schedules are view equivalent because Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

T1	**T2**
Read(A)	
	Write(A)

S1

T1	**T2**
	Write(A)
Read(A)	

S2

- **Updated Read:** In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.
- Below two schedules are not view equal because, in S1, T3 is reading A updated by T2 and in S2, T3 is reading A updated by T1.

T1	T2	T3
Write(A)		
	Write(A)	
		Read(A)

S1

T1	T2	T3
	Write(A)	
Write(A)		
		Read(A)

S2

- **Final Write:** A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.
- Below two schedules is view equal because Final write operation in S1 is done by T3 and in S2, the final write operation is also done by T3.

T1	T2	T3
Write(A)		
	Read(A)	
		Write(A)

S1

T1	T2	T3
	Read(A)	
Write(A)		
		Write(A)

S2

Example

- With 3 transactions, the total number of possible schedule are = 3! = 6

S1 = <T1 T2 T3>

S2 = <T1 T3 T2>

S3 = <T2 T3 T1>

S4 = <T2 T1 T3>

S5 = <T3 T1 T2>

S6 = <T3 T2 T1>

T1	T2	T3
Read(A)		
	Write(A)	
Write(A)		Write(A)

S1 = <T1 T2 T3>

Step 1: Final Updation on data items

- In both schedules S and S1, there is no read except the initial read that's why we don't need to check that condition.

Step 2: Initial Read

- The initial read operation in S is done by T1 and in S1, it is also done by T1.

Step 3: Final Write

- The final write operation in S is done by T3 and in S1, it is also done by T3. So, S and S1 are view Equivalent.
- The first schedule S1 satisfies all three conditions, so we don't need to check another schedule.

Hence, view equivalent serial schedule is: **T1 → T2 → T3**

T1	T2	T3
Read(A)		
	Write(A)	
Write(A)		Write(A)

S

T1	T2	T3
Read(A) Write(A)		
	Write(A)	
		Write(A)

S1

RECOVERABILITY

Unrecoverable Schedule

- The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit.

Example

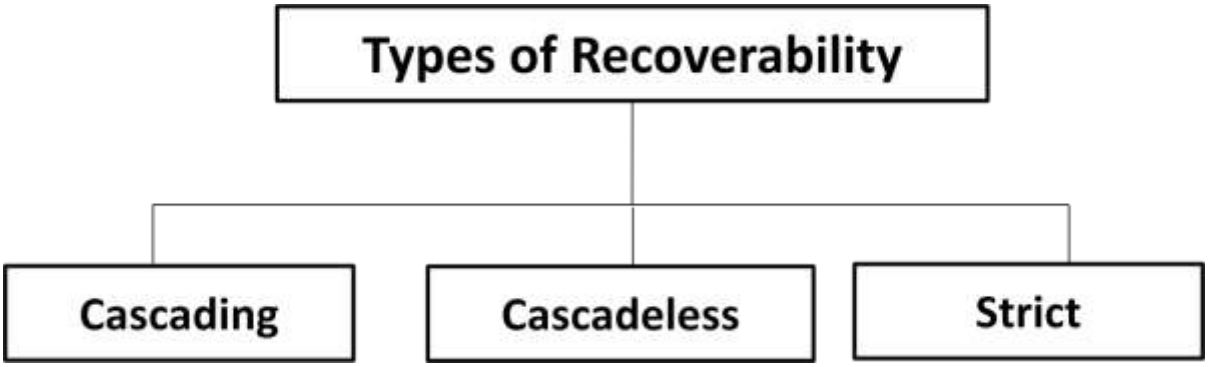
T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		
Failure Point				
Commit;				

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, but T2 can't be rollback because it already committed. So this type of schedule is known as irrecoverable schedule.

What is Recoverability?

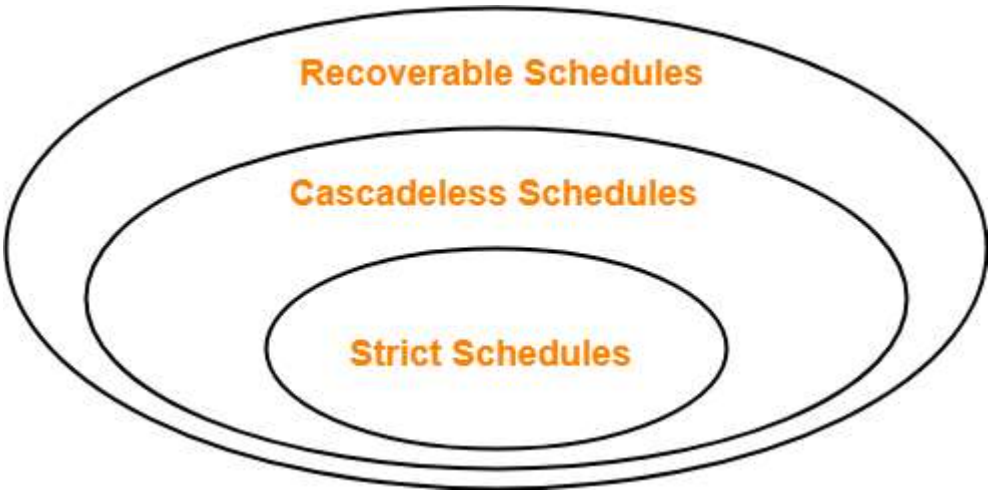
- Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback.
- But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.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

Types of Recoverability



Recoverable Schedule

- A schedule is said to be recoverable if it is recoverable as name suggest.
- Schedules in which transactions commit only after all transactions **whose changes they read commit** are called recoverable schedules.
- Only reads are allowed before write operation on same data.
- Only reads (Ti->Tj) is permissible.



Cascading (Recoverable with cascading rollback)

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

Example

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;		

- Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, 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.

Cascade less recoverable schedule

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.

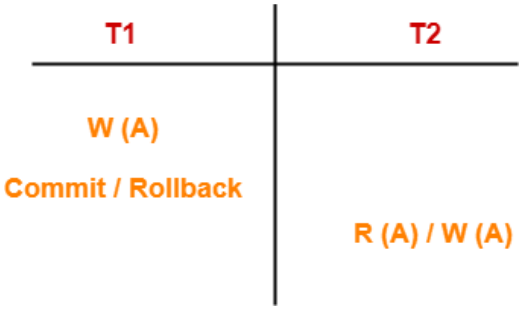
T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
Commit;		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		

Strict Schedule

- If schedule contains no **read** or **write** before commit then it is known as strict schedule. Strict schedule is strict in nature.
- Strict schedule allows only committed read and write operations.
- Clearly, strict schedule implements more restrictions than cascadeless schedule.

Example

- Strict schedules are more strict than cascadeless schedules.
- All strict schedules are cascadeless schedules.
- All cascadeless schedules are not strict schedules.



IMPLEMENTATION OF ISOLATION

There are various **concurrency-control** policies that we can use to ensure that, even when multiple transactions are executed concurrently, only acceptable schedules are generated, regardless of how the operating system time-shares resources (such as CPU time) among the transactions.

Lock

- A transaction acquires a **lock** on the entire database before it starts and releases the lock after it has committed.
- While a transaction holds a lock, no other transaction is allowed to acquire the lock, and all must therefore wait for the lock to be released.
- As a result of the locking policy, only one transaction can execute at a time.
- Therefore, only serial schedules are generated.
- These are trivially serializable, and it is easy to verify that they are recoverable and Cascadeless as well.
- A concurrency-control policy such as this one leads to poor performance, since it forces transactions to wait for preceding transactions to finish before they can start.

Goal of Concurrency Control

- The goal of concurrency-control policies is to provide a high degree of concurrency, while ensuring that all schedules that can be generated are conflict or view serializable, recoverable, and Cascadeless.

Concurrency control mechanisms - Locking

- Instead of locking the entire database, a transaction could, instead, lock only those data items that it accesses.
- Under such a policy, the transaction must hold locks long enough to ensure serializability, but for a period short enough not to harm performance excessively.

- Two-phase locking requires a transaction to have two phases, one where it acquires locks but does not release any, and a second phase where the transaction releases locks but does not acquire any.
- Further improvements to locking result if we have two kinds of locks:
    - **Shared**
    - **Exclusive**

Shared Lock

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

Exclusive Lock

- In the exclusive lock, the data item can be both reads as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

Timestamps

- Timestamps are used to ensure that transactions access each data item in order of the transactions‘ timestamps if their accesses conflict.
- When this is not possible, offending transactions are aborted and restarted with a new timestamp.

- Another category of techniques for the implementation of isolation assigns each transaction a **timestamp**, typically when it begins.
- For each data item, the system keeps two timestamps.
    - **Read Timestamp**
    - **Write Timestamp**

Read Timestamp

- The read timestamp of a data item holds the largest (that is, the most recent) timestamp of those transactions that read the data item.

Write Timestamp

- The write timestamp of a data item holds the timestamp of the transaction that wrote the current value of the data item.

Multiple Versions and Snapshot Isolation

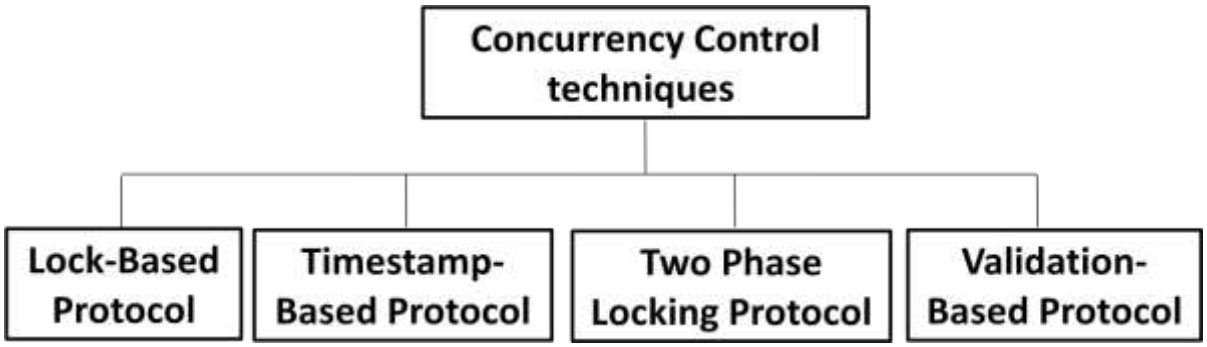
- By maintaining more than one version of a data item, it is possible to allow a transaction to read an old version of a data item rather than a newer version written by an uncommitted transaction or by a transaction that should come later in the serialization order.
- There are a variety of multi version concurrency control techniques. One in particular, called **snapshot isolation**, is widely used in practice.
- In snapshot isolation, we can imagine that each transaction is given its own version, or snapshot, of the database when it begins.
- It reads data from this private version and is thus isolated from the updates made by other transactions.
- If the transaction updates the database, that update appears only in its own version, not in the actual database itself.
- Information about these updates is saved so that the updates can be applied to the ―real‖ database if the transaction commits.
- When a transaction *T* enters the partially committed state, it then proceeds to the committed state only if no other concurrent transaction has modified data that *T* intends to update.
- Transactions that, as a result, cannot commit abort instead. Snapshot isolation ensures that attempts to read data never need to wait (unlike locking).
- Read-only transactions cannot be aborted; only those that modify data run a slight risk of aborting.
- Since each transaction reads its own version or snapshot of the database, reading data does not cause subsequent update attempts by other transactions to wait (unlike locking).
- Since most transactions are read-only (and most others read more data than they update), this is often a major source of performance improvement as compared to locking.

CONCURRENCY CONTROL

What is Concurrency Control?

- **Concurrency Control** in Database Management System is a procedure of managing simultaneous operations without conflicting with each other.
- It ensures that Database transactions are performed concurrently and accurately to produce correct results without violating data integrity of the respective Database.

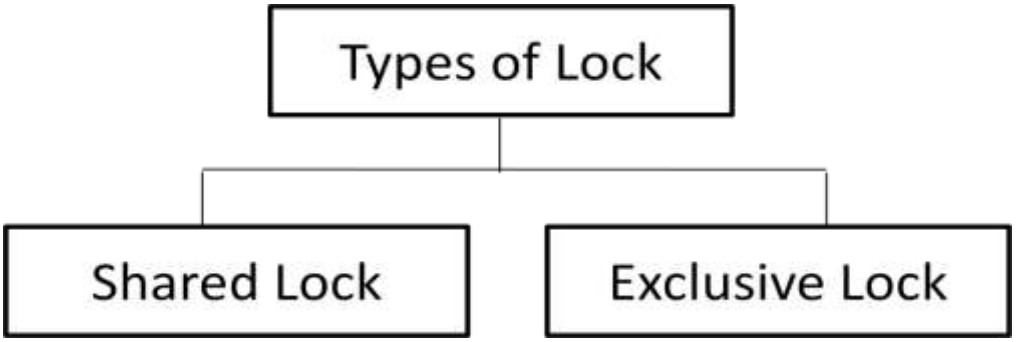
Concurrency Control techniques



LOCK-BASED PROTOCOL

- In this Lock-Based protocol, any transaction cannot read or write data until it acquires an appropriate lock on it.

Types of Lock



- **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*.
- **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*.

Concurrency Control Manager

- Every transaction **request** a lock in an appropriate mode on data item *Q*, depending on the types of operations that it will perform on *Q*.
- The transaction makes the request to the concurrency-control manager.
- The transaction can proceed with the operation only after the concurrency-control manager **grants** the lock to the transaction.
- The use of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

Compatibility function

- Given a set of lock modes, **compatibility function can be defined** on them as follows:

- Let *A* and *B* represent arbitrary lock modes. Suppose that a transaction *Ti* requests a lock of mode *A* on item *Q* on which transaction *Tj* (*Ti* = *Tj*) currently holds a lock of mode *B*.
- If transaction *Ti* can be granted a lock on *Q* immediately, in spite of the presence of the mode *B* lock, then we say mode *A* is **compatible** with mode *B*. Such a function can be represented conveniently by a matrix.

Lock-compatibility matrix comp

- The compatibility relation between the two modes of locking appears in the matrix comp.
- An element comp (*A*, *B*) of the matrix has the value *true* if and only if mode *A* is compatible with mode *B*.

	S	X
S	true	false
X	false	false

Compatibility function

- Shared mode is compatible with shared mode, but not with exclusive mode.
- At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item.
- A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released.

Lock

- A transaction requests a shared lock on data item *Q* by executing the lock S(*Q*) instruction.
- Similarly, a transaction requests an exclusive lock through the lock-X(*Q*) instruction.
- A transaction can unlock a data item *Q* by the unlock(*Q*) instruction. To access a data item, transaction *Ti* must first lock that item.
- If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released.
- Thus, *Ti* is made to **wait** until all incompatible locks held by other transactions have been released.

Example

Let *A* and *B* be two accounts that are accessed by transactions *T*1 and *T*2. Transaction *T*1 transfers $50 from account *B* to account *A:*

```
T1: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

Transaction *T*1

```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

Transaction *T*2

Transaction *T*2 displays the total amount of money in accounts *A* and *B*—that is, the sum *A* + B. Suppose that the values of accounts A and B are $100 and $200, respectively. If these two transactions are executed serially, either in the order T1, T2 or the order T2, T1, then transaction T2 will display the value $300. If, however, these transactions are executed concurrently, then schedule 1 is possible. In this case, transaction *T*2 displays $250, which is incorrect. The reason for this mistake is that the transaction *T*1 unlocked data item *B* too early, as a result of which *T*2 saw an inconsistent state.

T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

- The schedule shows the actions executed by the transactions, as well as the points at which the concurrency-control manager grants the locks.
- The transaction making a lock request cannot execute its next action until the concurrency control manager grants the lock.
- Hence, the lock must be granted in the interval of time between the lock-request operation and the following action of the transaction.
- Suppose now that unlocking is delayed to the end of the transaction.
- Transaction *T*3 corresponds to *T*1 with unlocking delayed.
- Transaction *T*4 corresponds to *T*2 with unlocking delayed.

```
T3: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(B);
    unlock(A).
```

Transaction *T*3 (transaction *T*1 with unlocking delayed)

- Suppose now that unlocking is delayed to the end of the transaction.
- Transaction *T*3 corresponds to *T*1 with unlocking delayed.
- Transaction *T*4 corresponds to *T*2 with unlocking delayed.

```
T4: lock-S(A);
    read(A);
    lock-S(B);
    read(B);
    display(A + B);
    unlock(A);
    unlock(B).
```

Transaction *T*4 (transaction *T*2 with unlocking delayed)

- The sequence of reads and writes in schedule 1, which lead to an incorrect total of $250 being displayed, is no longer possible with *T*3 and *T*4.
- Other schedules are possible.
- *T*4 will not print out an inconsistent result in any of them.

Deadlock

- Unfortunately, locking can lead to an undesirable situation.
- Consider the partial schedule for *T*3 and *T*4.
- Since *T*3 is holding an exclusive mode lock on *B* and *T*4 is requesting a shared-mode lock on *B*, *T*4 is waiting for *T*3 to unlock *B*.
- Similarly, since *T*4 is holding a shared-mode lock on *A* and *T*3 is requesting an exclusive-mode lock on *A*, *T*3 is waiting for *T*4 to unlock *A*.
- Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called **deadlock**.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- When deadlock occurs, the system must roll back one of the two transactions.

- Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked.
- These data items are then available to the other transaction, which can continue with its execution.

Locking Protocol

- Each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items.
- Locking protocols restrict the number of possible schedules.
- The set of all such schedules is a proper subset of all possible serializable schedules.

Terminologies

- Let *{T*0, *T*1, . . . , *Tn}* be a set of transactions participating in a schedule *S*.
- Say that *Ti* **precedes** *Tj* in *S*, written *Ti* → *Tj*, if there exists a data item *Q* such that *Ti* has held lock mode *A* on *Q*, and *Tj* has held lock mode *B* on *Q* later, and comp(*A*,*B*) = false.
- If *Ti* → *Tj*, then that precedence implies that in any equivalent serial schedule, *Ti* must appear before *Tj*.
- A schedule *S* is **legal** under a given locking protocol if *S* is a possible schedule for a set of transactions that follows the rules of the locking protocol.
- Say that a locking protocol **ensures** conflict serializability if and only if all legal schedules are conflict serializable; in other words, for all legal schedules the associated → relation is acyclic.

Granting of Locks

- When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted.

Starving

- Suppose a transaction *T*2 has a **shared-mode lock** on a data item, and another transaction *T*1 requests an **exclusive-mode lock** on the data item.
- Clearly, *T*1 has to wait for *T*2 to release the shared-mode lock.
- Meanwhile, a transaction *T*3 may request a shared-mode lock on the same data item.
- The lock request is compatible with the lock granted to *T*2, so *T*3 may be granted the shared-mode lock.

- At this point *T*2 may release the lock, but still *T*1 has to wait for *T*3 to finish.
- But again, there may be a new transaction *T*4 that requests a shared-mode lock on the same data item, and is granted the lock before *T*3 releases it.

- In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but *T*1 **never gets the exclusive-mode lock** on the data item.
- The transaction *T*1 may never make progress, and is said to be **starved**.

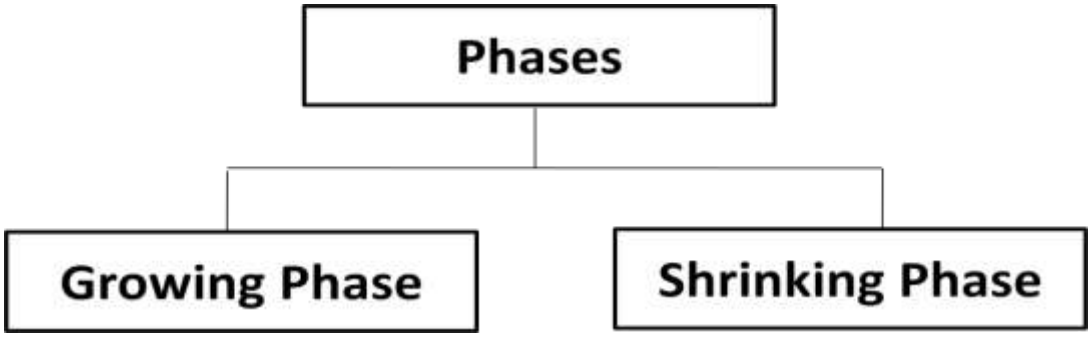
Solution to Starving

- When a transaction *Ti* requests a lock on a data item *Q* in a particular mode *M*, the concurrency-control manager grants the lock provided that:
  - There is no other transaction holding a lock on *Q* in a mode that conflicts with *M*.
  - There is no other transaction that is waiting for a lock on *Q* and that made its lock request before *Ti*.
- Thus, a lock request will never get blocked by a lock request that is made later.

THE TWO-PHASE LOCKING 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 phase.

Phases of Two-Phase Locking Protocol

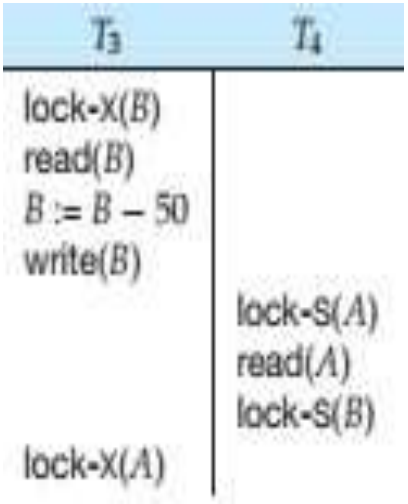


- **Growing phase**. A transaction may obtain locks, but may not release any lock.
- **Shrinking phase**. A transaction may release locks, but may not obtain any new locks.
- Initially, a transaction is in the **growing phase**.
- The transaction **acquires locks** as needed.

- Once the transaction **releases a lock**, it enters the **shrinking phase**, and it can issue no more lock requests.

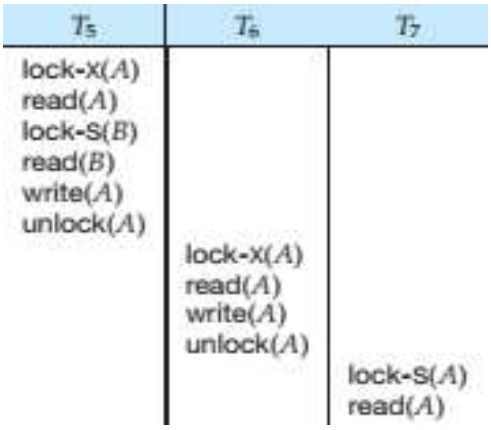
Example

- Transactions *T*3 and *T*4 are two phase.
- On the other hand, transactions *T*1 and *T*2 are not two phase.
- Note that the unlock instructions do not need to appear at the end of the transaction.
- For example, in the case of transaction *T*3, we could move the unlock(*B*) instruction to just after the lock-X(*A*) instruction, and still retain the two-phase locking property.

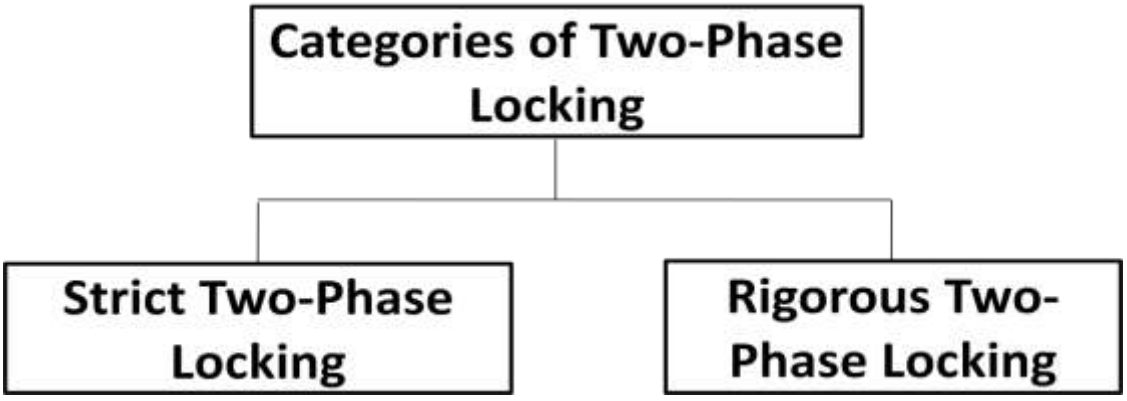


- Consider any transaction. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point** of the transaction.
- Now, transactions can be ordered according to their lock points— this ordering is, in fact, a serializability ordering for the transactions.
- Two-phase locking does *not* ensure freedom from deadlock.
- Observe that transactions *T*3 and *T*4 are two phase, but, in schedule 2 , they are deadlocked.

- Cascading rollback may occur under two-phase locking.
- Consider the partial schedule .Each transaction observes the two-phase locking protocol, but the failure of *T*5 after the read(A) step of *T* 7 leads to cascading rollback of *T*6 and *T* 7.



Categories of Two-Phase Locking Protocol



Strict Two-Phase Locking

- Cascading rollbacks can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol**.
- This protocol requires not only that locking be two phase, but also that all **exclusive-mode locks taken by a transaction be held until that transaction commits.**
- This requirement ensures that any data **written by an uncommitted transaction are locked in exclusive mode** until the transaction commits, preventing any other transaction from reading the data.

Rigorous Two-Phase Locking

- Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which requires that all locks be held until the transaction commits.

Example

- Consider the following two transactions, for which we have shown only some of the significant read and write operations:

T_8: read(a_1);
read(a_2);
...
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display($a_1 + a_2$).

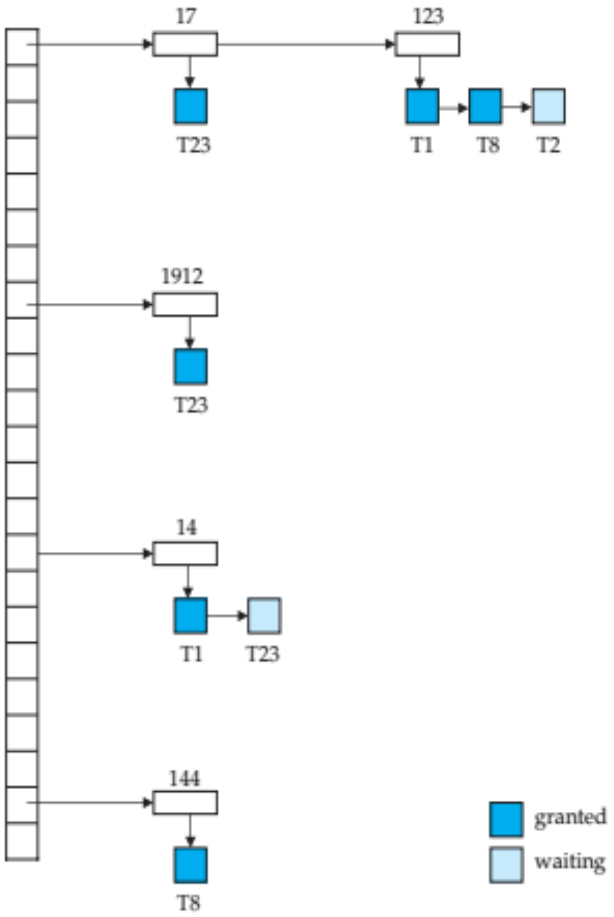
- If we employ the two-phase locking protocol, then *T*8 must lock *a*1 in exclusive mode.
- Therefore, any concurrent execution of both transactions amounts to a serial execution.
- Notice, however, that *T*8 needs an exclusive lock on *a*1 only at the end of its execution, when it writes *a*1.
- Thus, if *T*8 could initially lock *a*1 in shared mode, and then could later change the lock to exclusive mode, we could get more concurrency, since *T*8 and *T*9 could access *a*1 and *a*2 simultaneously.
- This observation leads us to a refinement of the basic two-phase locking protocol, in which **lock conversions** are allowed.
- A mechanism can be provided for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock.
- The conversion from shared to exclusive modes is done by **upgrade**, and from exclusive to shared by **downgrade**.
- Lock conversion cannot be allowed arbitrarily.
- Rather, **upgrading** can take place in only **the growing phase**, whereas **downgrading** can take place in only the **shrinking phase**.
- A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction, on the basis of read and write requests from the transaction:
- When a transaction *Ti* issues a read(*Q*) operation, the system issues a lockS(*Q*) instruction followed by the read(*Q*) instruction.

- When *Ti* issues a write(*Q*) operation, the system checks to see whether *Ti* already holds a shared lock on *Q*. If it does, then the system issues an upgrade(*Q*) instruction, followed by the write(*Q*) instruction.
- Otherwise, the system issues a lock-X(*Q*) instruction, followed by the write(*Q*) instruction.
- All locks obtained by a transaction are unlocked after that transaction commits or aborts.

Implementation of Locking

- A **lock manager** can be **implemented as a process** that **receives messages** from transactions and **sends messages** in reply.
- The lock-manager **process replies** to lock-request messages with lock-grant messages, or with messages requesting rollback of the transaction (in case of deadlocks).
- Unlock messages require only an acknowledgment in response, but may result in a grant message to another waiting transaction.
- The lock manager uses this data structure:
- For each data item that is currently locked, it maintains **a linked list** of records, one for each request, in the order in which the requests arrived.
- It uses **a hash table**, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the **lock table**.
- Each record of the linked list for a data item notes **which transaction** made the request, and **what lock mode** it requested.
- The record also notes if the request has currently been granted.

Lock Table



- The table contains locks for five different data items, I4, I7, I23, I44, and I912.
- The lock table uses **overflow chaining**, so there is a linked list of data items for each entry in the lock table.
- There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items.
- Granted locks are the rectangles filled in a darker shade, while waiting requests are the rectangles filled in a lighter shade.
- T23 has been granted locks on I912 and I7, and is waiting for a lock on I4.

Process of request by lock manager

- When a lock request message arrives, it adds a record to the end of the linked list for the data item, if the linked list is present. Otherwise it creates a new linked list, containing only the record for the request.
- It always grants a lock request on a data item that is not currently locked.
- But if the transaction requests a lock on an item on which a lock is currently held, the lock manager grants the request only if it is compatible with the locks that are currently held, and all earlier requests have been granted already.
- Otherwise the request has to wait.

- When the lock manager receives an **unlock message** from a transaction, it **deletes the record for that data item in the linked list corresponding to that transaction.**
- It tests the record that follows, if any, as described in the previous paragraph, to see if that request can now be granted.
- If it can, the lock manager grants that request, and processes the record following it, if any, similarly, and so on.
- If a transaction **aborts**, the lock manager **deletes any waiting request** made by the transaction.
- Once the database system has taken appropriate actions to **undo** the transaction, it **releases all locks held by the aborted transaction.**

Graph-Based Protocols

- With the prior knowledge about the order in which the database items will be accessed, it is possible to construct locking protocols that are not two phase, but that, nevertheless, ensure conflict serializability.
- To acquire such prior knowledge, a partial ordering → is imposed on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items.
- If $d_i \rightarrow d_j$, then any transaction accessing both *di* and *d j* must access d_i before accessing d_j.
- This partial ordering may be the result of either the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control.

Partial Ordering

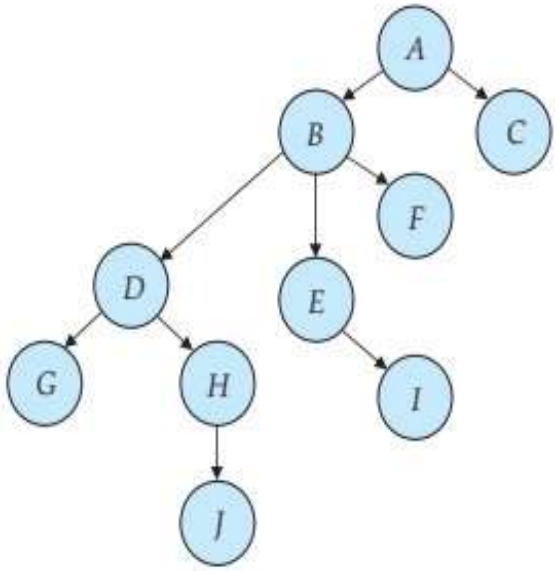
- The partial ordering implies that the set **D** may now be viewed as a directed acyclic graph, called a **database graph**.
- A simple protocol, called the **tree protocol**, which is restricted to employ only *exclusive* locks.

Tree Protocol

- In the **tree protocol**, the only lock instruction allowed is **lock-X**. Each transaction T_i can lock a data item at **most on**ce, and must observe the following rules:
  - **The first lock by T_i may be on any data item.**
  - **Subsequently, a data item *Q* can be locked by T_i only if the parent of *Q* is currently locked by T_i.**
  - **Data items may be unlocked at any time.**

– **A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i.**

Tree-Structured database graph



- The following four transactions follow the tree protocol on this graph.
- ***T*10:** lock-X(*B*); lock-X(*E*); lock-X(*D*); unlock(*B*); unlock(*E*); lock-X(*G*); unlock(*D*); unlock(*G*).
- ***T*11:** lock-X(*D*); lock-X(*H*); unlock(*D*); unlock(*H*).
- ***T*12:** lock-X(*B*); lock-X(*E*); unlock(*E*); unlock(*B*).
- ***T*13:** lock-X(*D*); lock-X(*H*); unlock(*D*); unlock(*H*).

Serializable schedule under the tree protocol

- *T*10: lock-X(*B*);
- lock-X(*E*); lock-X(*D*); unlock(*B*); unlock(*E*); lock-X(*G*); unlock(*D*); unlock(*G*).
- *T*11: lock-X(*D*); lock-X(*H*); unlock(*D*); unlock(*H*).
- *T*12: lock-X(*B*); lock-X(*E*); unlock(*E*); unlock(*B*).
- *T*13: lock-X(*D*); lock-X(*H*); unlock(*D*); unlock(*H*).

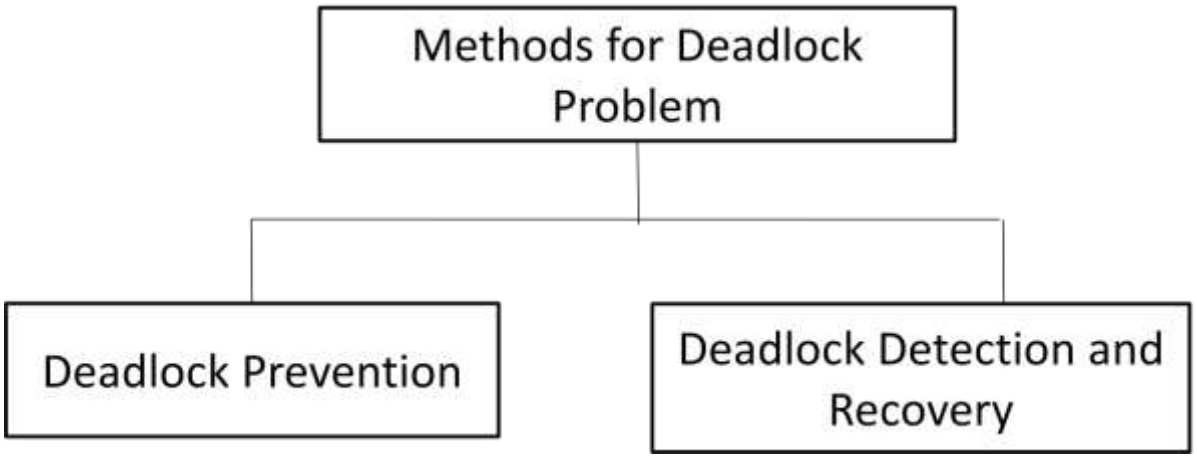
T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock(G)			

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.

Methods for Deadlock Problem



Deadlock prevention

- It ensures that the system will *never* enter a deadlock state.
- Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high.

Approaches to Deadlock Prevention

- One approach ensures that **no cyclic waits** can occur by ordering the requests for locks, or requiring all locks to be acquired together.
- The other approach is closer to deadlock recovery, and performs **transaction rollback instead of waiting for a lock**, whenever the wait could potentially result in a deadlock.

Approach 1

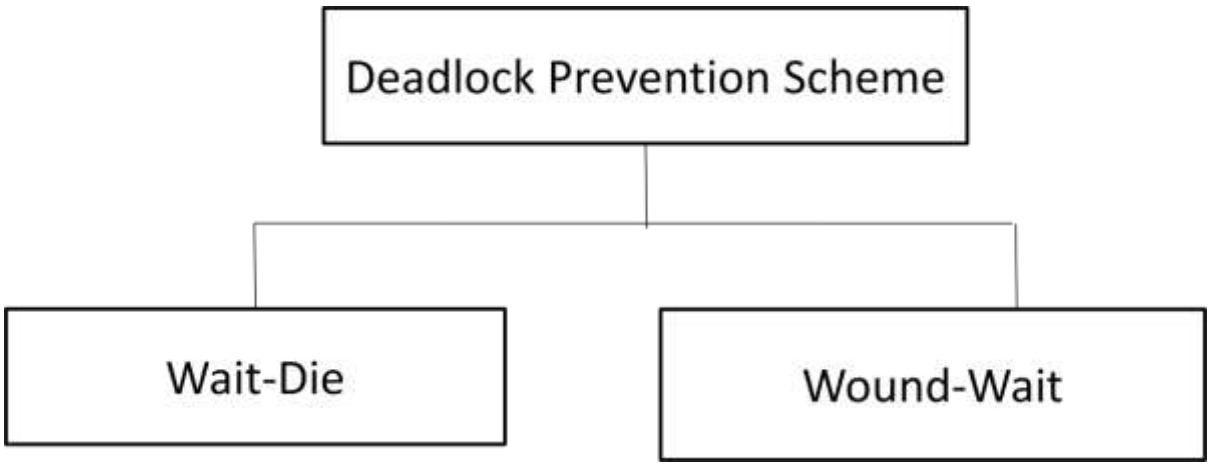
- Each transaction locks all its data items before it begins execution.
- Moreover, either **all are locked** in one step or **none are locked**.
- There are two main disadvantages to this protocol:
  - **it is often hard to predict, before the transaction begins, what data items need to be locked;**
  - **data-item utilization may be very low, since many of the data items may be locked but unused for a long time.**
- For preventing deadlocks is to impose an ordering of all data items, and to require that a transaction lock data items only in a sequence consistent with the ordering.
- Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the ordering.
- This scheme is easy to implement, as long as the set of data items accessed by a transaction is known when the transaction starts execution.
- There is no need to change the underlying concurrency-control system if two-phase locking is used: All that is needed is to ensure that locks are requested in the right order.

Approach 2

- For preventing deadlocks is to use preemption and transaction rollbacks.
- In preemption, when a transaction T_j requests a lock that transaction T_i holds, the lock granted to T_i may be **preempted** by rolling back of T_i, and granting of the lock to T_j.
- To control the preemption, we assign a unique timestamp, based on a counter or on the system clock, to each transaction when it begins.
- The system uses these timestamps only to decide whether a transaction should wait or roll back.

- Locking is still used for concurrency control.
- If a transaction is rolled back, it retains its *old* timestamp when restarted.

Deadlock Prevention Scheme



Wait-Die

- The **wait–die** scheme is a **non preemptive technique**.
- When transaction T_i requests a data item currently held by T_j, T_i is allowed to wait only if it has a timestamp **smaller than** that of T_j (that is, *Ti* is older than T_j).
- Otherwise, T_i is rolled back (dies).

  For example, suppose that transactions *T*14, *T*15, and *T*16 have timestamps 5, 10, and 15, respectively. If *T*14 requests a data item held by *T*15, then *T*14 will wait. If *T*24 requests a data item held by *T*15, then *T*16 will be rolled back

Wound-wait

- The **wound–wait** scheme is a preemptive technique.
- It is a counterpart to the wait–die scheme.
- When transaction T_i requests a data item currently held by T_j, T_i is allowed to wait only if it has a timestamp **larger than** that of T_j (that is, T_i is younger than T_j).
- Otherwise, T_j is rolled back (T_j is *wounded* by T_i)

Problem

Unnecessary rollbacks may occur

Lock timeouts

- In this approach, a transaction that has requested a lock waits for at most a specified amount of time.
- If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts.

- If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed.
- This scheme falls somewhere between deadlock prevention, where a deadlock will never occur, and deadlock detection and recovery.

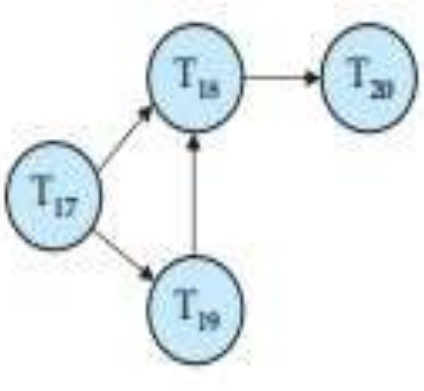
Deadlock Detection and Recovery

- The system is allowed to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme.
- An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred.
- If one has, then the system must attempt to recover from the deadlock.
- To do so, the system must
    - Maintain **information about the current allocation of data items** to transactions, as well as any outstanding data item requests.
    - Provide an algorithm that uses this information to **determine whether the system has entered a deadlock state**.
    - **Recover from the deadlock** when the detection algorithm determines that a deadlock exists.

Deadlock Detection

- Deadlocks can be described precisely in terms of a directed graph called a **wait-for graph**.
- 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.
- Each element in the set E of edges 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 transaction T_i to T_j, implying that transaction T_i is waiting for transaction T_j 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 in the system if and only if the wait-for graph contains a cycle.
- Each transaction involved in the cycle is said to be deadlocked.

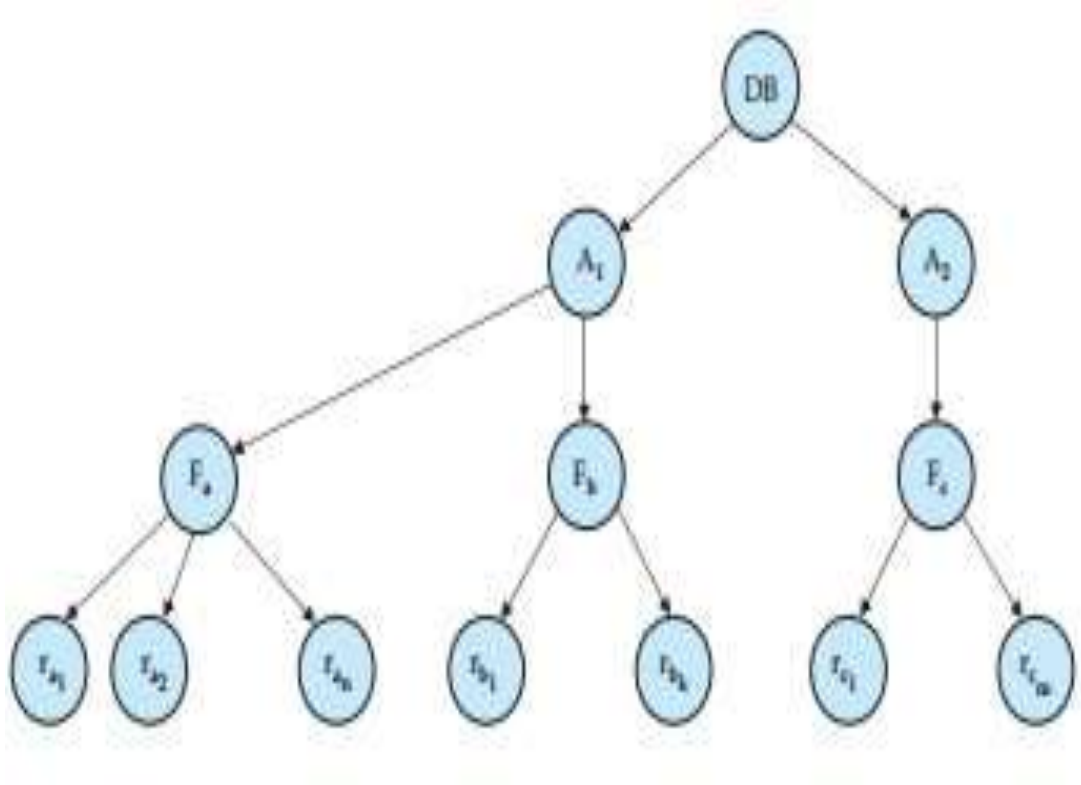
- To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.



- Transaction T_{17} is waiting for transactions T_{18} and T_{19}.
- Transaction T_{19} is waiting for transaction T_{18}.
- Transaction T_{18} is waiting for transaction T_{20}.
- Since the graph has no cycle, the system is not in a deadlock state.
- Suppose now that transaction *T*20 is requesting an item held by *T*19. The edge *T*20 → *T*19 is added to the wait-for graph, resulting in the new system state in Figure 15.14. This time, the graph contains the cycle:
  *T*18 → *T*20 → *T*19 → *T*18
  implying that transactions *T*18, *T*19, and *T*20 are all deadlocked

Multiple Granularity

- **Granularity –** It is the **size of the data** item allowed to lock.
- *Multiple Granularity* means **hierarchically breaking up the database into blocks** that can be locked and can be tracked needs what needs to lock and in what fashion.
- Such a hierarchy can be represented graphically as a tree.
- Each node in the tree can be locked individually.
- As in the 2-phase locking protocol, it shall use shared and exclusive lock modes.



- When a transaction locks a node, in either shared or exclusive mode, the transaction also implicitly locks all the descendants of that node in the same lock mode.

  For example, if transaction T_i gets an explicit lock on file F_c in exclusive mode, then it has an implicit lock in exclusive mode on all the records belonging to that file. It does not need to lock the individual records of F_c explicitly.

Intention Mode Lock

- In addition to **S** and **X** lock modes, there are three additional lock modes with multiple granularities:
  - **Intention-Shared (IS):** explicit locking at a lower level of the tree but only with shared locks.
  - **Intention-Exclusive (IX):** explicit locking at a lower level with exclusive or shared locks.
  - **Shared & Intention-Exclusive (SIX):** the sub tree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive mode locks.

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Compatibility Matrix

Multiple-granularity locking protocol

- The **multiple-granularity locking protocol** uses these lock modes to ensure serializability. It requires that a transaction *Ti* that attempts to lock a node *Q* must follow these rules:
  - **Transaction *Ti* must observe the lock-compatibility function.**
  - **Transaction *Ti* must lock the root of the tree first, and can lock it in any mode.**
  - **Transaction *Ti* can lock a node *Q* in S or IS mode only if *Ti* currently has the parent of *Q* locked in either IX or IS mode.**
  - **Transaction *Ti* can lock a node *Q* in X, SIX, or IX mode only if *Ti* currently has the parent of *Q* locked in either IX or SIX mode.**
  - **Transaction *Ti* can lock a node only if *Ti* has not previously unlocked any node (that is, *Ti* is two phase).**
  - **Transaction *Ti* can unlock a node *Q* only if *Ti* currently has none of the children of *Q* locked.**

TIMESTAMP-BASED PROTOCOLS

- Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a *timestamp-ordering* scheme.

Timestamps

- With each transaction *Ti* in the system, a unique fixed timestamp is associated, 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*).
- The timestamps of the transactions determine the serializability order. Thus, if TS(*Ti*) < TS(*Tj*), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction *Ti* appears before transaction *Tj*.
- 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.

Methods

1. Use the value of the **system clock** as the timestamp; that is, a transaction‘s timestamp is equal to the value of the clock when the transaction enters the system.
2. Use a **logical counter** that is incremented after a new timestamp has been assigned; that is, a transaction‘s timestamp is equal to the value of the counter when the transaction enters the system.

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*).

3. 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*).

Example

***T*25:** read(*B*);

read(*A*);

display(*A* + *B*).

Transaction *T*25 displays the contents of accounts *A* and *B*.

***T*26:** read(*B*);

B := *B* - 50;

write(*B*);

read(*A*);

A := *A* + 50;

write(*A*);

display(*A* + *B*)

Transaction *T*26 transfers $50 from account *B* to account *A*, and then displays the contents of both

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

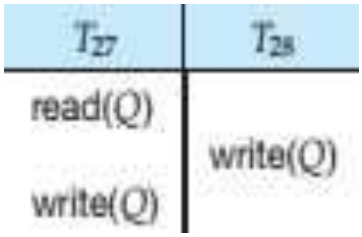
TS(*T*25) < TS(*T*26)

- Whenever a schedule does not follow a serializability order according to the Timestamp, a user generally rejects it and rollback the Transaction.
- Some operations on the other hand are harmless and can be allowed.

Thomas Write Rule

- Thomas Write Rule provides the guarantee of serializability order for the protocol. It improves the Basic Timestamp Ordering Algorithm.
- The basic Thomas write rules are as follows:
  - If TS(T) < R_TS(X) then transaction T is aborted and rolled back, and operation is rejected.

- If TS(T) < W_TS(X) then don't execute the W_item(X) operation of the transaction and continue processing.
- If neither condition 1 nor condition 2 occurs, then allowed to execute the WRITE operation by transaction Ti and set W_TS(X) to TS(T).



A Serializable Schedule that is not Conflict Serializable

VALIDATION-BASED PROTOCOLS

- Validation phase is also known as optimistic concurrency control technique. In the validation based protocol, the transaction is executed in the following three phases:
  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 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.

Three Timestamps

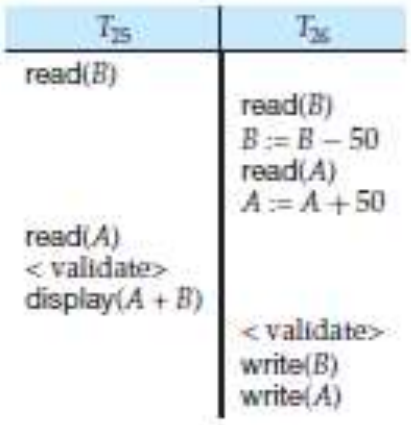
1. **Start**(*Ti*), the time when *Ti* started its execution.
2. **Validation**(*Ti*), the time when *Ti* finished its read phase and started its validation phase.
3. **Finish**(*Ti*), the time when *Ti* finished its write phase.

The serializability order is determined 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.



Schedule 6, a schedule produced by using validation

Asan illustration, consider again transactions T25 and T26. Suppose that TS(T25) < TS(T26). Then, the validation phase succeeds in the schedule 6 in Figure 15.19. Note that thewrites to the actual variables are performed only after the validation phase of T26. Thus, T25 reads the old values of B and A, and this schedule is serializable.

The validation scheme automatically guards against cascading rollbacks, since the actual writes take place only after the transaction issuing the write has committed. However, there is a possibility of starvation of long transactions, due to a sequence of conflicting short transactions that cause repeated restarts of the long transaction. To avoid starvation, conflicting transactions must be temporarily blocked, to enable the long transaction to finish. 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 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.

RECOVERY AND ATOMICITY

What is Recovery System?

- A **recovery scheme** that can restore the database to the consistent state that existed before the failure.
- The recovery scheme must also provide **high availability**; that is, it must minimize the time for which the database is not usable after a failure.

Failure Classification

- **Transaction failure**. There are two types of errors that may cause a transaction to fail:
  - **Logical error**. The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
  - **System error**. The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be reexecuted at a later time.
- **System crash**. There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted. The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the **fail-stop assumption**. Well-designed systems have numerous internal checks, at the hardware and the software level, that bring the system to a halt when there is an error.
- **Disk failure**. A disk block loses its content as a result of either a head crash or failure during a data-transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as DVD or tapes, are used to recover from the failure.

Recover from Failure

- To determine how the system should recover from failures
  - Identify the failure modes of those devices used for storing data
  - Consider how these failure modes affect the contents of the database.

Parts of Recovery Algorithms

- Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.
- Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability

Storage Structure

- **Volatile storage** − As the name suggests, a volatile storage cannot survive system crashes. Volatile storage devices are placed very close to the CPU; normally they are embedded onto the chipset itself.

For example, main memory and cache memory are examples of volatile storage. They are fast but can store only a small amount of information.

- **Non-volatile storage** − These memories are made to survive system crashes. They are huge in data storage capacity, but slower in accessibility. Examples may include hard-disks, magnetic tapes, flash memory, and non-volatile (battery backed up) RAM.
- Block transfer between memory and disk storage can result in:
  - **Successful completion**. The transferred information arrived safely at its destination.
  - **Partial failure**. A failure occurred in the midst of transfer, and the destination block has incorrect information.
  - **Total failure**. The failure occurred sufficiently early during the transfer that the destination block remains intact.

Recovery

- If a data-transfer failure occurs, the system detects it and invokes a recovery procedure to restore the block to a consistent state.
- To do so, the system must maintain two physical blocks for each logical database block; in the case of mirrored disks, both blocks are at the same location; in the case of remote backup, one of the blocks is local, whereas the other is at a remote site.
- An output operation is executed as follows:
  - **1.** Write the information onto the first physical block.

    2. When the first write completes successfully, write the same information onto the second physical block.

    3. The output is completed only after the second write completes successfully

    If the system fails while blocks are being written, it is possible that the two copies of a block are inconsistent with each other.

- During recovery, for each block, the system would need to examine two copies of the blocks. If both are the same and no detectable error exists, then no further actions are necessary.
- If the system detects an error in one block, then it replaces its content with the content of the other block. If both blocks contain no detectable error, but they differ in content, then the system replaces the content of the first block with the value of the second. This recovery procedure ensures that a write to stable storage either succeeds completely (that is, updates all copies) or results in no change.

Data Access

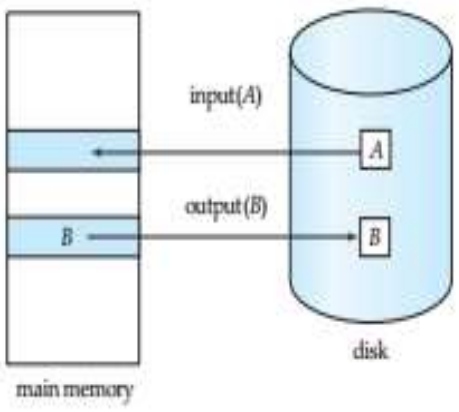
- The database is partitioned into fixed-length storage units called **blocks**.
- Blocks are the units of data transfer to and from disk, and may contain several data items.

Blocks

- Transactions input information from the disk to main memory, and then output the information back onto the disk.
- The input and output operations are done in block units.
- The blocks residing on the disk are referred to as **physical blocks**; the blocks residing temporarily in main memory are referred to as **buffer blocks**. The area of memory where blocks reside temporarily is called the **disk buffer**.
- Block movements between disk and main memory are initiated through the following two operations:

  1. input(*B*) transfers the physical block *B* to main memory.

  2. output(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.



Block Storage Operation

Log

- The most widely used structure for recording database modifications is the **log.**

- The log is a sequence of **log records, recording all the update activities in the** database.

Update log record

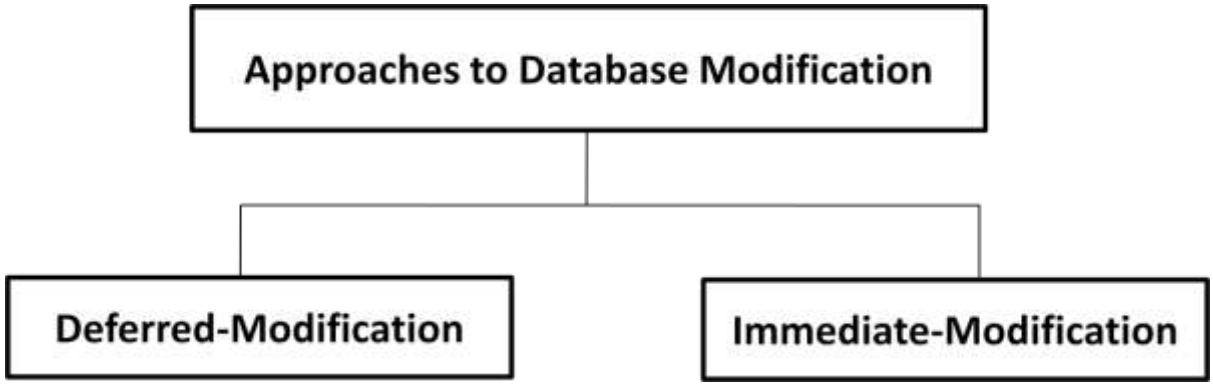
- An **update log record describes a single** database write.
- It has these fields:
  - **Transaction identifier**
  - **Data-item identifier**
  - **Old value**
  - **New value**
- **Transaction identifier, which is the unique identifier of the transaction that** performed the write operation.
- **Data-item identifier, which is the unique identifier of the data item written.** Typically, it is the location on disk of the data item, consisting of the block identifier of the block on which the data item resides, and an offset within the block.
- **Old value**, which is the value of the data item prior to the write.
- **New value**, which is the value that the data item will have after the write.
- An update log record is represented **as<*Ti , Xj , V1, V2*>**, *indicating that transaction Ti has performed a write on data item Xj . Xj had value V1 before the write, and* has value *V2 after the write.*

- Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction.
- Among the types of log records are:
  - ***<T_i start>. Transaction T_i has started.***
  - ***<T_i commit>. Transaction T_i has committed.***
  - ***<T_i abort>. Transaction T_i has aborted.***
- Whenever a transaction performs a write, it is essential that the log record for that write be created and added to the log, before the database is modified.
- Once a log record exists, we can output the modification to the database if that is desirable.
- Also, we have the ability to *undo a modification that has already been* output to the database.
- We undo it by using the old-value field in log records.
- For log records to be useful for recovery from system and disk failures, the log must reside in stable storage.
- Every log record is written to the end of the log on stable storage as soon as it is created.

Database Modification

- A transaction creates a log record prior to modifying the database.
- The log records allow the system to **undo changes** made by a transaction in the event that the transaction must be **aborted**; they allow the system also to **redo changes** made by a transaction if the transaction **has committed** but the **system crashed before those changes could be stored in the database** on disk.
- The system is allowed to perform *undo and redo operations as appropriate.*
  - **Undo using a log record sets the data item specified in the log record to the old value.**
  - **Redo using a log record sets the data item specified in the log record to the new value.**
- The steps a transaction takes in modifying a data item:
  - **The transaction performs some computations in its own private part of main memory.**
  - **The transaction modifies the data block in the disk buffer in main memory holding the data item.**
  - **The database system executes the output operation that writes the data block to disk.**
- A transaction ***modifies the database*** *if it performs* ***an update on a disk*** **buffer**, or on the disk itself; **updates to the private part** of main memory **do not count as database modifications.**

Approaches to Database Modification



Deferred-modification

- If a transaction **does not modify the database until it has committed**, it is said to use the **deferred-modification technique.**
- Deferred modification has the overhead that transactions need to **make local copies of all updated data items**;
- Further, if a transaction reads a data item that it has updated, it must read the value from its local copy. It is also called NO-UNDO/REDO technique.
- It is used for the recovery of the transaction failures which occur due to power, memory or OS failures.
- Whenever any transaction is executed, the updates are not made immediately to the database.
- They are first recorded on the log file and then those changes are applied once commit is done. This is called ―Re-doing‖ process.
- Once the rollback is done none of the changes are applied to the database and the changes in the log file are also discarded.
- If commit is done before crashing of the system, then after restarting of the system the changes that have been recorded in the log file are thus applied to the database.

Immediate-modification

- If database modifications occur while the **transaction is still active**, the transaction is said to use the **immediate-modification technique.**
- It is also called UNDO/REDO technique.
- Whenever any transaction is executed, the updates are made directly to the database and the log file is also maintained which contains both old and new values.
- Once **commit is done**, all the changes get **stored permanently** into the database and records in log file are thus discarded.
- Once **rollback is done** the **old values get restored** in the database and all the changes made to the database are also discarded. This is called **―Un-doing‖** process.
- If commit is done before crashing of the system, then after restarting of the system the changes are stored permanently in the database.

Deferred Update	**Immediate Update**
In deferred update, the changes are not applied immediately to the database.	In immediate update, the changes are applied directly to the database.

The log file contains all the changes that are to be applied to the database.	The log file contains both old as well as new values.
In this method once rollback is done all the records of log file are discarded and no changes are applied to the database.	In this method once rollback is done the old values are restored into the database using the records of the log file.
Concepts of buffering and caching are used in deferred update method.	Concept of shadow paging is used in immediate update method.
The major disadvantage of this method is that it requires a lot of time for recovery in case of system failure.	The major disadvantage of this method is that there are frequent I/O operations while the transaction is active.

RECOVERY WITH CONCURRENT TRANSACTIONS

- Concurrency control means that multiple transactions can be executed at the same time and then the interleaved logs occur.
- But there may be changes in transaction results so maintain the order of execution of those transactions.
- During recovery, it would be very difficult for the recovery system to backtrack all the logs and then start recovering.

Using the Log to Redo and Undo Transactions

Consider our simplified banking system. Let *T0 be a transaction that transfers* $50 from account *A to account B:*

<T_0 start>
<T_0, A, **1000**, 950>
<T_0, B, **2000**, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, **700**, 600>
<T_1 commit>
	T0: *read(A);*
	A := A − 50;
	write(*A*);
	read(*B*);
	B := B + 50;
	write(*B*)
		T1: *read(C);*
		C := C − 100;
		write(*C*)

A=1000

B=2000

C=700

Portion of the system log corresponding to T_0 and T_1

Let T1 be a transaction that withdraws $100 from account C

State of system log and database

Log	**Database**
<T_0 start> <T_0 , A, **1000**, 950> <T_0 , B**, 2000**, 2050> <T_0 commit> <T_1 start> <T_1 , C, **700**, 600> <T_1 commit>	A=950 B=2050 C=600

Log

<T0 start>	<T0 start>	<T0 start>
<T0 , A, 1000, 950>	<T0 , A, 1000, 950>	<T0 , A, 1000, 950>
<T0 , B, 2000, 2050>	<T0 , B, 2000, 2050>	<T0 , B, 2000, 2050>
	<T0 commit>	<T0 commit>
	<T1 start>	<T1 start>
	<T1 , C, 700, 600>	<T1 , C, 700, 600>
		<T1 commit>

- First, let us assume that the crash occurs just after the log record for the step:

  write(*B*)

  of transaction *T0 has been written to stable.*
- When the system comes back up, it finds the record <T0 start> in the log, but no corresponding <T0 commit> or <T0 abort> record.
- Thus, transaction T0 must be undone, so an undo(T0) is performed. As a result, the values in accounts A and B (on the disk) are restored to $1000 and $2000, respectively.
- Next, let us assume that the crash comes just after the log record for the step:

  write(*C*)

  of transaction *T1 has been written to stable storage.*
- When the system comes back up, two recovery actions need to be taken. The operation undo(T1) must be performed, since the record <T1 start> appears in the log, but there is no record <T1 commit> or <T1 abort>.

- The operation redo(T0) must ben performed, since the log contains both the record <T0 start> and the record <T0 commit>. At the end of the entire recovery procedure, the values of accounts A, B, and C are $950, $2050, and $700, respectively.
- Finally, let us assume that the crash occurs just after the log record:

  <T1 commit>

  has been written to stable storage.
- When the system comes back up, both T0 and T1 need to be redone, since the records <T0 start> and <T0 commit> appear in the log, as do the records<T1 start>and<T1 commit>.
- After the system performs the recovery procedures redo(T0) and redo(T1), the values in accounts A, B, and C are $950, $2050, and $600, respectively.

Checkpoints

- When a system crash occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone.
- In principle, we need to search the entire log to determine this information.

Difficulties with this approach

- The search process is **time-consuming**.
- Most of the transactions that, according to our algorithm, need to **be redone** have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer.
- To reduce these types of overhead, checkpoints are introduced.
- A simple checkpoint scheme that
    - **does not permit any updates** to be performed while the checkpoint operation is in progress, and
    - **outputs all modified buffer blocks to disk** when the checkpoint is performed.

What is Checkpoint?

- The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.
- The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked, and the transaction is executed then using the steps of the transaction, the log files will be created.

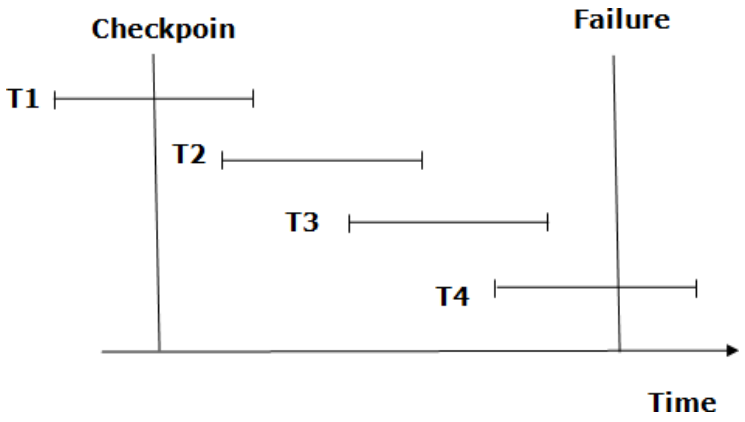
- When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.
- A checkpoint is performed as follows:
    - **Output onto stable storage all log records currently residing in main memory.**
    - **Output to the disk all modified buffer blocks.**
    - **Output onto stable storage a log record of the form <*checkpoint L*>**

Where L is a list of transactions active at the time of the checkpoint.

- Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress.
- The presence of a ***<checkpoint L>*** *record in the log allows the system to* streamline its recovery procedure. Consider a transaction *Ti that completed prior* to the checkpoint. For such a transaction, the *<Ti commit>record (or < Ti abort>* record) appears in the log before the *<checkpoint> record. Any database modifications* made by *Ti must have been written to the database either prior to the* checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on *Ti* .
- After a system crash has occurred, the system examines the log to find the last *<checkpoint L> record (this can be done by searching the log backward, from* the end of the log, until the first *<checkpoint L> record is found).*
- The redo or undo operations need to be applied only to transactions in *L, and* to all transactions that started execution after the *<checkpoint L> record was* written to the log. Let us denote this set of transactions as *T.*
- For all transactions T_k *in T that have no <*T_k *commit> record or <*T_k *abort>* record in the log, execute undo(T_k).
- For all transactions T_k *in T such that either the record <*T_k *commit> or the* record <T_k *abort> appears in the log, execute redo(*T_k *).*

Recovery using Checkpoints

- The recovery system reads log files from the end to start. It reads log files from T4 to T1.
- Recovery system maintains two lists, a **redo-list**, and an **undo-list**.
- The transaction is put into **redo state** if the recovery system sees a log with **<Tn, Start> and <Tn, Commit> or just <Tn, Commit>.**



- In the redo-list and their previous list, all the transactions are removed and then redone before saving their logs.

  For example: In the log file, transaction T2 and T3 will have <Tn, Start> and <Tn, Commit>. The T1 transaction will have only <Tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts T1, T2 and T3 transaction into redo list.

- The transaction is put into **undo state** if the recovery system sees a log with **<Tn, Start> but no commit or abort log found.** In the undo-list, all the transactions are undone, and their logs are removed.

  For example: Transaction T4 will have <Tn, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed amid.

Fuzzy Checkpoints

- The check pointing technique requires that all **updates to the database be temporarily suspended while the checkpoint is in progress.**
- If the **number of pages** in the buffer **is large**, a **checkpoint may take a long time to finish**, which can result in an unacceptable interruption in processing of transactions.
- To avoid such interruptions, the **check pointing technique can be modified** to permit updates to start once the checkpoint record has been written, but before the modified buffer blocks are written to disk.
- The checkpoint thus generated is a **fuzzy checkpoint.**
- It **reduces the time** it takes to checkpoint the database.
- Fuzzy check pointing is done as follows:

  1. **Temporarily stop all updates** by transactions

  2. **Write a log record and force log** to stable storage

  3. Note **list M of modified buffer blocks**

4. Now **permit transactions to proceed** with their actions
5. **Output to disk all modified buffer blocks** in list M
    - **blocks should not be updated while being output**
    - **Follow WAL: all log records pertaining to a block must be output before the block is output**
6. Store a pointer to the checkpoint record in a fixed position last_checkpoint on disk

- When recovering using a fuzzy checkpoint, start scan from the checkpoint record pointed to by last_checkpoint
    - **Log records before last_checkpoint have their updates reflected in database on disk, and need not be redone.**
    - **Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely**

Failure with loss of nonvolatile storage

- Technique similar to check pointing used to deal with loss of nonvolatile storage
- Periodically dump the entire content of the database to stable storage say, once per day.
- If a failure occurs that results in the loss of physical database blocks, the system uses the most recent dump in restoring the database to a previous consistent state.
- Once this restoration has been accomplished, the system uses the log to bring the database system to the most recent consistent state.
- One approach to database dumping requires that no transaction may be active during the dump procedure, and uses a procedure similar to check pointing:

1. Output all log records currently residing in main memory onto stable storage.

2. Output all buffer blocks onto the disk.

3. Copy the contents of the database to stable storage.

4.* Output a log record <*dump*> *onto the stable storage.

Recovering from Failure of Non-Volatile Storage

- To recover from the loss of nonvolatile storage, the system restores the database to disk by using the most recent dump.
- Then, it consults the log and redoes all the actions since the most recent dump occurred.
- No undo operations need to be executed.
- In case of a partial failure of nonvolatile storage, such as the failure of a single block or a few blocks, only those blocks need to be restored, and redo actions performed only for

those blocks. A dump of the database contents is also referred to as an **archival dump,** since we can archive the dumps and use them later to examine old states of the database.

- Dumps of a database and check pointing of buffers are similar.
- Most database systems also support an **SQL dump, which writes out SQL DDL** statements and **SQL insert statements to a file**, which can then be reexecuted to re-create the database.
- Such dumps are useful when migrating data to a different instance of the database, or to a different version of the database software, since the physical locations and layout may be different in the other database instance or database software version.

REMOTE BACKUP SYSTEMS

Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed. Traditional transaction-processing systems are centralized or client–server systems. Such systems are vulnerable to environmental disasters such as fire, flooding, or earthquakes. Increasingly, there is a need for transaction-processing systems that can function in spite of system failures or environmental disasters. Such systems must provide **high availability**; that is, the time for which the system is unusable must be extremely small.

We can achieve high availability by performing transaction processing at one site, called the **primary site**, and having a **remote backup** site where all the data from the primary site are replicated. The remote backup site is sometimes also called the **secondary site**. The remote site must be kept synchronized with the primary site, as updates are performed at the primary. We achieve synchronization by sending all log records from primary site to the remote backup site. The remote backup site must be physically separated from the primary—for example, we can locate it in a different state—so that a disaster at the primary does not damage the remote backup site. Figure shows the architecture of a remote backup system.

When the primary site fails, the remote backup site takes over processing. First, however, it performs recovery, using its (perhaps outdated) copy of the data from the primary, and the log records received from the primary. In effect, the remote backup site is performing recovery actions that would have been performed at the primary site when the latter recovered. Standard recovery algorithms, with minor modifications, can be used for recovery at the remote backup site. Once recovery has been performed, the remote backup site starts processing transactions.

Availability is greatly increased over a single-site system, since the system can recover even if all data at the primary site are lost. Several issues must be addressed in designing a remote backup system:

- **Detection of failure:** Backup site must detect when primary site has failed
  - to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.

- **Transfer of control:**
  - To take over control backup site first perform recovery using its copy of the database and all the long records it has received from the primary.
    - Thus, completed transactions are redone and incomplete transactions are rolled back.
  - When the backup site takes over processing it becomes the new primary
  - To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.
- **Time to recover:** To reduce delay in takeover, backup site periodically processes the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.
- Hot-Spare configuration permits very fast takeover:
  - Backup continually processes redo log record as they arrive, applying the updates locally.
  - When failure of the primary is detected the backup rolls back incomplete transactions, and is ready to process new transactions.
- **Time to commit**. To ensure that the updates of a committed transaction are durable, a transaction must not be declared committed until its log records have reached the backup site. This delay can result in a longer wait to commit a transaction, and some systems therefore permit lower degrees of durability. The degrees of durability can be classified as follows:

- Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.
  - **One-safe**
  - **Two-very-safe**
  - **Two-safe**

- **One-safe:** commit as soon as transaction‘s commit log record is written at primary **Problem:** updates may not arrive at backup before it takes over.
- **Two-very-safe:** commit when transaction‘s commit log record is written at primary and backup. Reduces availability since transactions cannot commit if either site fails.
- **Two-safe:** proceed as in two-very-safe if both primary and backup are active.
- If only the primary is active, the transaction commits as soon as is commit log record is written at the primary.
- Better availability than two-very-safe; avoids problem of lost transactions in one-safe.

ARIES

- ARIES stands for ―Algorithm for Recovery and Isolation Exploiting Semantics‖.
- ARIES uses logs to record the progress of transactions and their actions which cause changes to recoverable data objects.
- The log is the source of truth and is used to ensure that committed actions are reflected in the database, and that uncommitted actions are undone.
- Conceptually the log is a single ever-growing sequential file (append-only).
- Every log record has a unique log sequence number (LSN), and LSNs are assigned in ascending order.
- Log records are first written to volatile storage (e.g. in-memory), and at certain times – such as transaction commit – the log records up to a certain point (LSN) are written to stable storage. This is known as *forcing* the log up to that LSN.
- A system may periodically force the log buffers as they fill up.
- ARIES is able to avoid redoing many logged operations that have already been applied and to reduce the amount of information logged.
- ARIES uses
    - **Log Sequence Number (LSN) :** to identify log records, and the use of LSNs in database pages to identify which operations have been applied to a database page.
    - **Physiological Redo:**
    - **Dirty Page Table**
    - **Fuzzy Check pointing**

Data Structures in ARIES

- ARIES uses

  - **Log Sequence Number (LSN):** Each page on disk has pageLSN. LSN of the last log entry for that page
  - **Transaction table :** Each entry has lastLSN. Transaction table tracks all active transactions
  - **Dirty page table :** Each entry has recoveryLSN. Dirty page table tracks all dirty pages
- ARIES splits a log into multiple log files, each of which has a file number.
- When a log file grows to some limit, ARIES appends further log records to a new log file; the new log file has a file number that is higher by 1 than the previous log file.

PageLSN

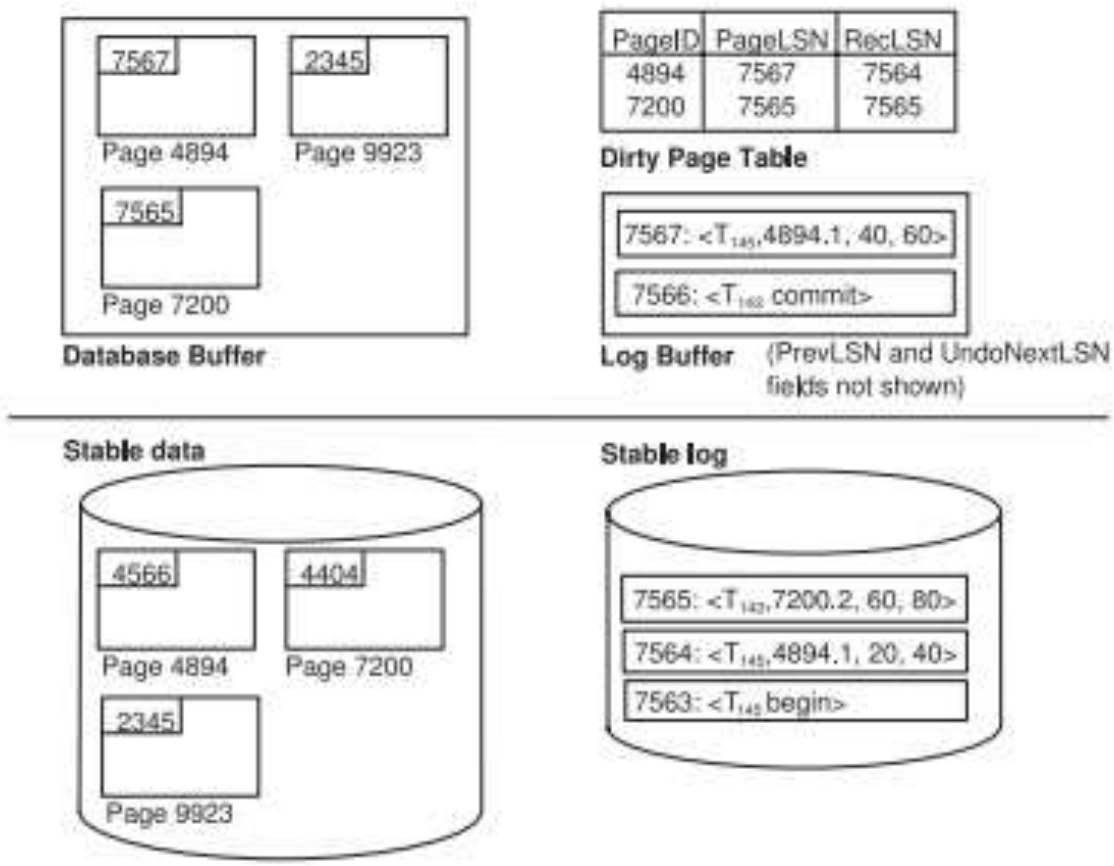
- The LSN then consists of a file number and an offset within the file.
- Each page also maintains an identifier called the **PageLSN**.
- Whenever an operation occurs on a page, the operation stores the LSN of its log record in the PageLSN field of the page.
- During the redo phase of recovery, any log records with LSN less than or equal to the PageLSN of a page should not be executed on the page, since their actions are already reflected on the page.

Each log record in ARIES has a log sequence number (LSN) that uniquely identifies the record. The number is conceptually just a logical identifier whose value is greater for log records that occur later in the log. In practice, the LSN is generated in such a way that it can also be used to locate the log record on disk. Typically,nARIES splits a log into multiple log files, each of which has a file number. When a log file grows to some limit, ARIES appends further log records to a new log file; the new log file has a file number that is higher by 1 than the previous log file. The LSN then consists of a file number and an offset within the file. Each page also maintains an identifier called the **PageLSN**. Whenever an update operation (whether physical or physiological) occurs on a page, the operation stores the LSN of its log record in the PageLSN field of the page.

During the redo phase of recovery, any log records with LSN less than or equal to the PageLSN of a page should not be executed on the page, since their actions are already reflected on the page. In combination with a scheme for recording PageLSNs as part of checkpointing, which we present later, ARIES can avoid even reading many pages for which logged operations are already reflected on disk. Thereby, recovery time is reduced significantly. The PageLSN is essential for ensuring idempotence in the presence of physiological redo operations, since reapplying a physiological redo that has already been

applied to a page could cause incorrect changes to a page. Pages should not be flushed to disk while an update is in progress, since physiological operations cannot be redone on the partially updated state of the page on disk. Therefore, ARIES uses latches on buffer pages to prevent them from being written to disk while they are being updated. It releases the buffer page latch only after the update is completed, and the log record for the update has been written to the log. Each log record also contains the LSN of the previous log record of the same transaction. This value, stored in the PrevLSN field, permits log records of a transaction to be fetched backward, without reading the whole log. There are special redo-only log records generated during transaction rollback, called **compensation log records** (**CLR**s) in ARIES. These serve the same purpose as the redo-only log records in our earlier recovery scheme. In addition CLRs serve the role of the operation-abort log records in our scheme. The CLRs have an extra field, called the UndoNextLSN, that records the LSN of the log that needs to be undone next, when the transaction is being rolled back. This field serves the same purpose as the operation identifier in the operation-abort log record in our earlier recovery scheme, which helps to skip over log records that have already been rolled back. The DirtyPageTable contains a list of pages that have been updated in the database buffer. For each page, it stores the PageLSN and a field called the RecLSN, which helps identify log records that have been applied already to the version of the page on disk. When a page is inserted into the DirtyPageTable (when it is first modified in the buffer pool), the value of RecLSN is set to the current end of log. Whenever the page is flushed to disk, the page is removed from the DirtyPageTable.

A checkpoint log record contains the DirtyPageTable and a list of active transactions. For each transaction, the checkpoint log record also notes LastLSN, the LSN of the last log record written by the transaction. A fixed position on disk also notes the LSN of the last (complete) checkpoint log record. Figure illustrates some of the data structures used in ARIES. The log records shown in the figure are prefixed by their LSN; these may not be explicitly stored, but inferred from the position in the log, in an actual implementation. The data item identifier in a log record is shown in two parts, for example 4894.1; the first identifies the page, and the second part identifies a record within the page (we assume a slotted page record organization within a page).



Data structures used in ARIES

Each page (whether in the buffer or on disk) has an associated PageLSN field. You can verify that the LSN for the last log record that updated page 4894 is 7567. By comparing PageLSNs for the pages in the buffer with the PageLSNs for the corresponding pages in stable storage, you can observe that the DirtyPageTable contains entries for all pages in the buffer that have been modified since they were fetched from stable storage. The RecLSN entry in the DirtyPageTable reflects the LSN at the end of the log when the page was added to DirtyPageTable, and would be greater than or equal to the PageLSN for that page on stable storage.

Recovery Algorithm

ARIES recovers from a system crash in three passes.

• **Analysis pass:** This pass determines which transactions to undo, which pages were dirty at the time of the crash, and the LSN from which the redo pass should start.

• **Redo pass:** This pass starts from a position determined during analysis, and performs a redo, repeating history, to bring the database to a state it was in before the crash.

• **Undo pass:** This pass rolls back all transactions that were incomplete at the time of crash.

Analysis Pass

The analysis pass finds the last complete checkpoint log record, and reads in the DirtyPageTable from this record. It then sets RedoLSN to the minimum of the RecLSNs of the pages in the DirtyPageTable. If there are no dirty pages, it sets RedoLSN to the LSN of the checkpoint log record. The redo pass starts its scan of the log from RedoLSN. All the log records earlier than this point have already been applied to the database pages on disk. The analysis pass initially sets the list of transactions to be undone, undo-list, to the list of transactions in the checkpoint log record. The analysis pass also reads from the checkpoint log record the LSNs of the last log record for each transaction in undo-list. The analysis pass continues scanning forward from the checkpoint. Whenever it finds a log record for a transaction not in the undo-list, it adds the transaction to undo-list. Whenever it finds a transaction end log record, it deletes the transaction from undo-list. All transactions left in undo-list at the end of analysis have to be rolled back later, in the undo pass. The analysis pass also keeps track of the last record of each transaction in undo-list, which is used in the undo pass. The analysis pass also updates DirtyPageTable whenever it finds a log record for an update on a page. If the page is not in DirtyPageTable, the analysis pass adds it to DirtyPageTable, and sets the RecLSN of the page to the LSN of the log record.

Redo Pass

The redo pass repeats history by replaying every action that is not already reflected in the page on disk. The redo pass scans the log forward from RedoLSN. Whenever it finds an update log record, it takes this action:

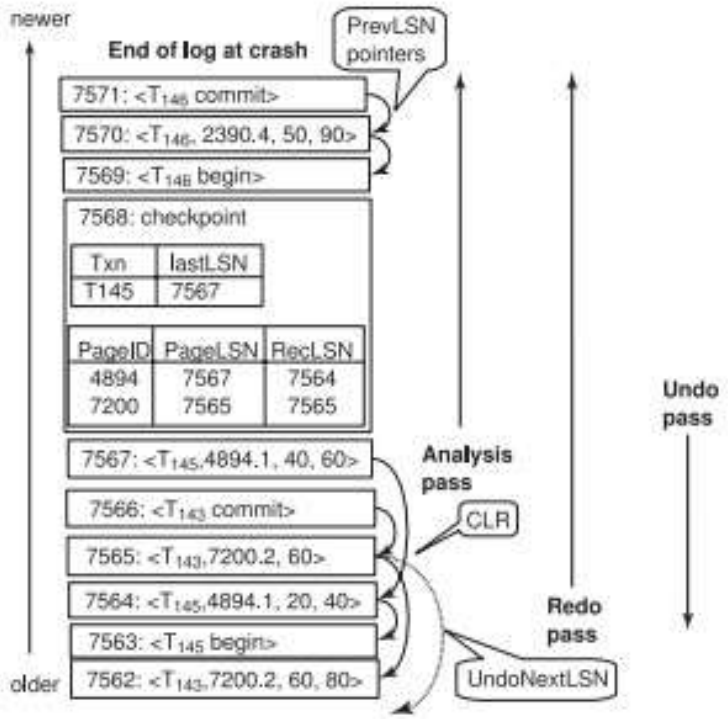
1. If the page is not in DirtyPageTable or the LSN of the update log record is less than the RecLSN of the page in DirtyPageTable, then the redo pass skips the log record.
2. Otherwise the redo pass fetches the page from disk, and if the PageLSN is less than the LSN of the log record, it redoes the log record.

If either of the tests is negative, then the effects of the log record have already appeared on the page; otherwise the effects of the log record are not reflected on the page. Since ARIES allows non-idempotent physiological log records, a log record should not be redone if its effect is already reflected on the page. If the first test is negative, it is not even necessary to fetch the page from disk to check its PageLSN.

Undo Pass and Transaction Rollback

The undo pass is relatively straightforward. It performs a single backward scan of the log, undoing all transactions in undo-list. The undo pass examines only log records of transactions in undo-list; the last LSN recorded during the analysis pass is used to find the last log record for each transaction in undo-list. Whenever an update log record is found, it is used to

perform an undo (whether for transaction rollback during normal processing, or during the restart undo pass). The undo pass generates a CLR containing the undo action performed (which must be physiological). It sets the UndoNextLSN of the CLR to the PrevLSN value of the update log record. If a CLR is found, its UndoNextLSN value indicates the LSN of the next log record to be undone for that transaction; later log records for that transaction have already been rolled back. For log records other than CLRs, the PrevLSN field of the log record indicates the LSN of the next log record to be undone for that transaction. The next log record to be processed at each stop in the undo pass is the maximum, across all transactions in undo-list, of next log record LSN.



Recovery actions in ARIES

Figure illustrates the recovery actions performed by ARIES, on an example log. Assume that the last completed checkpoint pointer on disk points to the checkpoint log record with LSN 7568. The PrevLSN values in the log records are shown using arrows in the figure, while the UndoNextLSN value is shown using a dashed arrow for the one compensation log record, with LSN 7565, in the figure. The analysis pass would start from LSN 7568, and when it is complete, RedoLSN would be 7564. Thus, the redo pass must start at the log record with LSN 7564. LSN is less than the LSN of the checkpoint log record, since the ARIES checkpointing algorithm does not flush modified pages to stable storage. The DirtyPageTable at the end of analysis would include pages 4894, 7200 from the checkpoint log record, and

2390 which is updated by the log record with LSN 7570. At the end of the analysis pass, the list of transactions to be undone consists of only T145 in this example.

The redo pass for the above example starts from LSN 7564 and performs redo of log records whose pages appear in DirtyPageTable. The undo pass needs to undo only transaction T145, and hence starts from its LastLSN value 7567, and continues backwards until the record < T145 start> is found at LSN 7563.