**DATABASE MANAGEMENT SYSTEMS**
**TRANSACTION CONCEPT AND STATE**

## THE CONCEPT OF A TRANSACTION:

A transaction is a set of operations used to perform a logical unit of work. It is a unit of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data- manipulation language or programming language.

Transactions access data using two operations:

- ***read(X)***, which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.

- ***write(X)***, which transfers the data item X from the local buffer of the transaction that executed the write back to the database.

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

$T_i$: read($A$);
$A := A - 50$;
write($A$);
read($B$);
$B := B + 50$;
write($B$).

## ACID Properties:

The acronym ACID is sometimes used to refer to the four properties of transactions that we have presented here: atomicity, consistency, isolation and durability. These ensure tomaintain data in the face of concurrent access and system failures:

**Atomicity:** 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. Examples of such failures include power failures, hardware failures, and software errors. 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. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction Ti is executed to completion, there exists a point at which the value of account A is $950 and the value of account B is $2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account A is $950, and the value of account B is$2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: 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, and, if the transaction does not complete its execution, the database system restores the old values to make it appear as though the transaction never executed.

Ensuring atomicity is the responsibility of the database system itself; specifically, it is handled by a component called the transaction-management component.

**Consistency:** The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction. Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction.

**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.

Ex: the database is temporarily inconsistent while the transaction to transfer funds from A to B is executing, with the deducted total written to A and the increased total yet to be written to

B. If a second concurrently running transaction reads A and B at this intermediate point and computes A+B, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on A and B based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other. However, concurrent execution of transactions provides significant performance benefits, as they allow multiple transactions to execute concurrently. The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order. Ensuring the isolation property is the responsibility of a component of the database system called the concurrency-control component.

**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 will 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. We 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. We can guarantee durability by ensuring that either

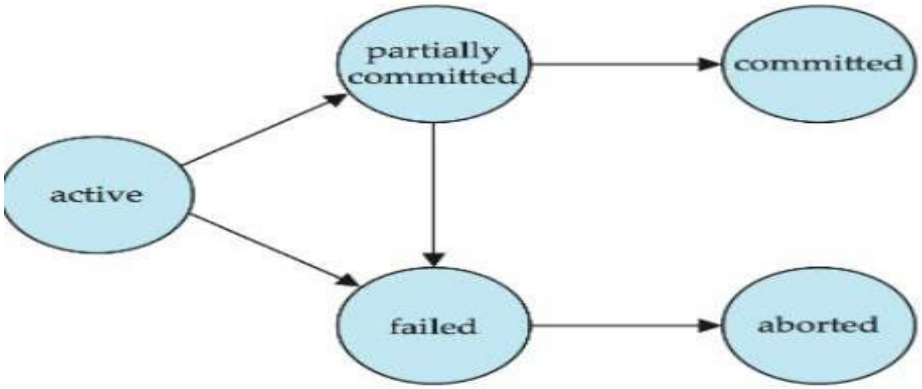
1. The updates carried out by the transaction have been written to disk before the transaction completes.

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.

Ensuring durability is the responsibility of a component of the database system called the recovery-management component.

## Transaction State

**Transaction State Diagram: A** simple abstract transaction model is shown in fig below:




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.

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. As mentioned earlier, we assume for now that failures do not result in loss of data on disk.

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.

# Implementation of Atomicity and Durability

Atomicity and durability are ACID properties that ensure the reliability and consistency of transactions in DBMS

**Atomicity:**

It is one of the key characteristics of transactions in database management systems (DBMS), which guarantees that every operation within a transaction is handled as a single, indivisible unit of work.

Either all the changes made by a transaction are committed to the database or none of them are committed.

**Durability:**

Durability, guarantees that changes made by a transaction once it has been committed are permanently kept in the database and will not be lost even in the event of a system failure or crash.

The recovery-management component of a database system can support atomicity and durability by a variety of schemes. We first consider a simple, but extremely inefficient, scheme called the *shadow copy scheme.*

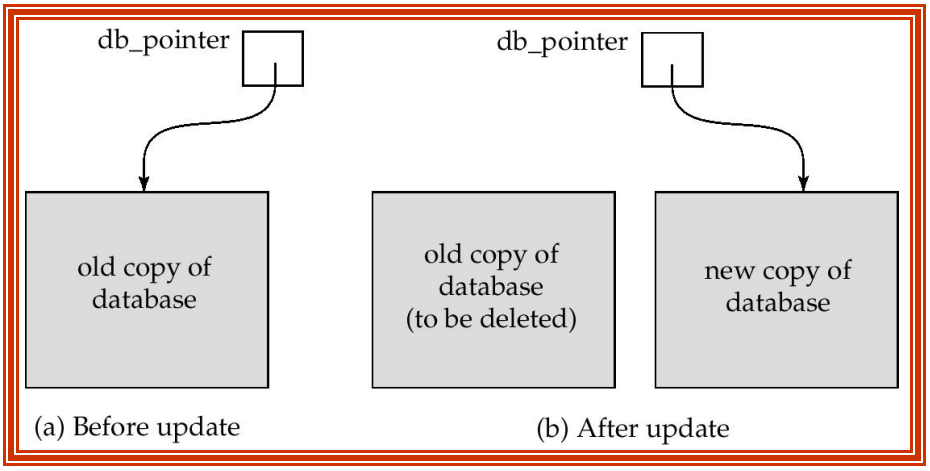
In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.

This scheme is based on making copies of the database, called shadow copies, assumes that only one transaction is active at a time. The scheme also assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk; it points to the current copy of the database.

If the transaction completes, it is committed as follows:

First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk. (Unix systems use the flush command for this purpose.)
After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database. The old copy of the database is then deleted.

The transaction is said to have been committed at the point where the updated db-pointer is written to disk.




We now consider how the technique handles transaction and system failures.

## Maintaining Automicity

- First, consider transaction failure. If the transaction fails at any time before db-pointer is updated, the old contents of the database are not affected.
- We can abort the transaction by just deleting the new copy of the database.
- Once the transaction has been committed, all the updates that it performed are in the database pointed to by db-pointer.
- Thus, either all updates of the transaction are reflected, or none of the effects are reflected, regardless of transaction failure.

## Maintaining Durability

- Suppose that the system fails at any time before the updated db-pointer is written to disk. Then, when the system restarts, it will read db- pointer and will thus see the original contents of the database, and none of the effects of the transaction will be visible on the database.
- Next, suppose that the system fails after db- pointer has been updated on disk. Before the pointer is updated, all updated pages of the new copy of the database were written to disk.
- Again, we assume that, once a file is written to disk, its contents will not be damaged even if there is a system failure. Therefore, when the system restarts, it will read db-pointer and will thus see the contents of the database after all the updates performed by the transaction.

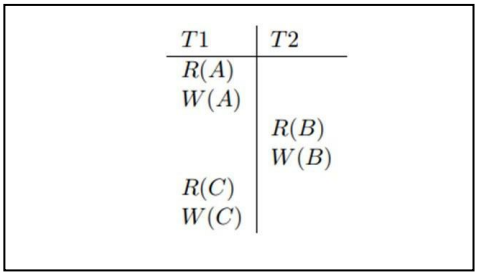
# Concurrent Executions

## Transaction and Schedule:

A **transaction** is seen by the DBMS as a series, or list, of actions. The actions that can be executed by a transaction include reads and writes of database objects.

In addition to reading and writing, each transaction must specify as its final action either commit (i.e., complete successfully) or abort (i.e., terminate and undo all the actions carried out thus far).

A **schedule** is a list of actions (reading, writing, aborting, or committing) from a set of transactions, i.e. a schedule is a chronological sequence of execution of one or more transactions.




Sample Schedule

A schedule that contains either an abort or a commit for each transaction whose actions are listed in it is called a **complete schedule.**

A schedule can be of two types:

1. Serial Schedule
2. Concurrent Schedule

In **Serial schedule**, transactions will be executed one after other, even the actions of different transactions are not interleaved.

In a **Concurrent Schedule** the actions of different transactions are interleaved to improve performance.

## Concurrent Executions

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data. However, there are two good reasons for allowing concurrency:

• ***Improved throughput and resource utilization:***

- A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU.

- The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction.
- All of this increases the throughput of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk utilization also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

• ***Reduced waiting time:***

- There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction.
- If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them.
- Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the average response time: the average time for a transaction to be completed after it has been submitted.

Let T1 and T2 be two transactions that transfer funds from one account to another. Transaction T1 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 T2 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 T1followed by T2.This execution sequence appears in Figure below. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T1 appearing in the left column and instructions of T2 appearing in the right column. The final values of accounts A and B, after the execution in Figure below, 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($B$) | |
| | read($A$) |
| | $temp := A * 0.1$ |
| | $A := A - temp$ |
| | write($A$) |
| | read($B$) |
| | $B := B + temp$ |
| | write($B$) |

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

Similarly, if the transactions are executed one at a time in the order T2 followed by T1, then the corresponding execution sequence is that of Figure below. 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($B$) |
| read($A$) | |
| $A := A - 50$ | |
| write($A$) | |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |

Schedule 2—a serial schedule in which $T_2$ is followed by $T_1$.

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.

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

Schedule 3—a concurrent schedule equivalent to schedule 1.

Not all concurrent executions result in a correct state. To illustrate, consider the schedule of Figure below:

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

Schedule 4—a concurrent schedule.

After the execution of this schedule, we arrive at a state where the final values of accounts A and B are $950 and $2100, respectively. This final state is an inconsistent state, since we have gained $50 in the process of the concurrent execution. Indeed, the sum A + B is not preserved by the execution of the two transactions.

If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state, such as the one just described, are possible. It is the job of the database system to ensure that any schedule that gets executed will leave the database in a consistent state. The concurrency-control component of the database system carries out this task.

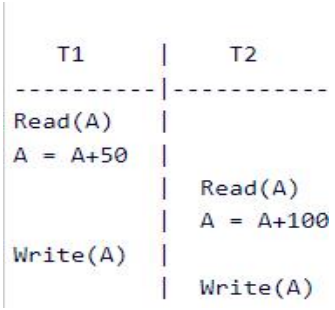
## Potential Problems with Concurrent Execution:

1. Lost Update Problems(W-W Conflict)
2. Uncommitted Data / Dirty read problem(W-R Conflict)
3. Unrepeatable data / Inconsistent retrievals(W-R Conflict)

**Lost Update Problems(W-W Conflict)**

Lost update problem occurs when two or more transactions modify the same data, resulting in the update being overwritten or lost by another transaction.

**Examples:**

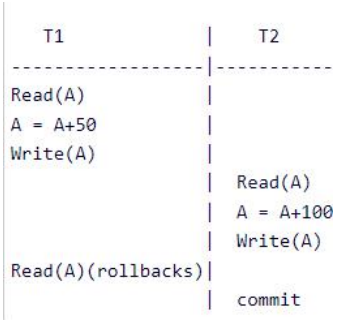



**Result:** T1's updates are lost.

**Uncommitted Data / Dirty read problem(W-R Conflict)**

The dirty read problem occurs when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.

One transaction might read an inconsistent state of data that's being updated by another.

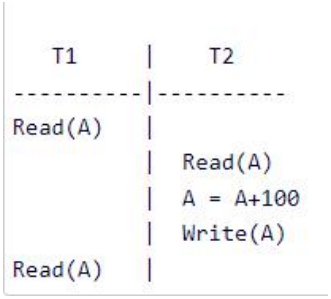



**Result:** T2 has a "dirty" value, that was never committed in T1 and doesn't actually exist in the database.

**Unrepeatable data / Inconsistent retrievals(W-R Conflict)**

Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

When a single transaction reads the same row multiple times and observes different values each time. This occurs because another concurrent transaction has modified the row between the two reads.




**Result:** Within the same transaction, T1 has read two different values for the same data item. This inconsistency is the unrepeatable read.

# Serializability Part I

## Transaction and Schedule:

A serializable schedule over a set S of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S. That is, the database instance that results from executing the given schedule is identical to the database instance that results from executing the transactions in some serial order.

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

if these actions are 'swapped' to obtain the serial schedule Tl; T2.

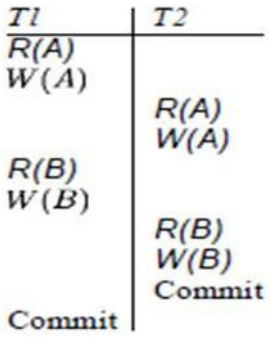



Figure 16.2 A Serializable Schedule

Two forms of serializabilty are

- **Conflict Serializabilty**
- **View Serializability.**

## Conflict Serializability:

Let us consider a schedule S in which there are two consecutive instructions Ii and Ij ,of transactions Ti and Tj, respectively (i ≠ j). If Ii and Ij refer to different data items, then we can swap Ii and Ij without affecting the results of any instruction in the schedule. However, if Ii and Ij refer to the same data item Q, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider

1. Ii = read(Q), Ij = read(Q). The order of Ii and Ij does not matter, since the same value of Q is read by Ti and Tj, regardless of the order.

2. Ii= read(Q), Ij= write(Q).If Ii comes before Ij, then Ti does not read the value of Q that is written by Tj in instruction Ij. If Ij comes before Ii, then Ti reads the value of Q that is written by Tj. Thus , the order of Ii and Ij matters.

3. Ii= write(Q), Ij= read(Q). the order of Ii and Ij matters for reasons similar to those of the previous case.

4. Ii= write(Q), Ij= 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 Ii and Ij in S, then the order of Ii and Ij directly affects the final value of Q in the database state that results from schedule S.

Thus, only in the case where both Ii and Ij are read instructions does the relative order of their execution not matter.

We say that Ii and Ij conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation. To illustrate the concept of conflicting instructions, we consider schedule 3, in Fig above. The write(A) instruction of T1 conflicts with the read(A) instruction of T2. However, the write(A) instruction of T2 does not conflict with the read(B) instruction of T1, because the two instructions access different data items.

Let Ii and Ij be consecutive instructions of a schedule S. If Ii and Ij are instructions of different transactions and Ii and Ij do not conflict, then we can swap the order of Ii and Ij to produce a new schedule S'. We expect S to be equivalent to S'. Since all instructions appear in the same order in both schedules except for Ii and Ij, whose order does not matter.

Since the write (A) instruction of Tin schedule 3 does not conflict with the read (B) instruction of T1, we can swap these instructions to generate an equivalent schedule, schedule 5,shown in Figure below.

| $T_1$ | $T_2$ |
|---|---|
| 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.

Regardless of the initial system state, schedules 3 and 5 both produce the same final system state. We continue to swap nonconflicting instructions:

- Swap the read(B) instruction of T1 with the read(A) instruction of T2
- Swap the write(B) instruction of T1 with the write(A) instruction of T2
- Swap the write(B) instruction of T1 with the read(A) instruction of T2.

The final result of these swaps, schedule 6 of Figure below, is a serial schedule

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| write($A$) | |
| read($B$) | |
| write($B$) | |
| | read($A$) |
| | write($A$) |
| | read($B$) |
| | write($B$) |

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

This equivalence implies that, regardless of the initial system state, schedule 3 will produce the same final state as will some serial schedule.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.

Finally, consider schedule 7 of Figure below; it consists of only the significant operations (that is, the read and write)of transactions T3 and T4. This schedule is not conflict serializable, since it is not equivalent to either the serial schedule <T3,T4> or the serial schedule <T4,T3>. It is possible to have two schedules that produce the same outcome, but that are not conflict equivalent.

| $T_3$ | $T_4$ |
|---|---|
| read($Q$) | |
| | write($Q$) |
| write($Q$) | |

**Precedence Graph for Testing Conflict Serializability in DBMS**

A Precedence Graph or Serialization Graph is used commonly to test the Conflict Serializability of a schedule. It is a directed Graph (V, E) consisting of a set of nodes V = {T1, T2, T3… Tn} and a set of directed edges E = {e1, e2, e3…..em}.

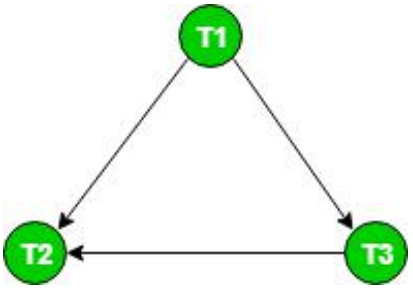
Consider some schedule of a set of transactions T1, T2, , Tn.

Precedence graph — a directed graph where

- the vertices are the transactions (names).
- there is an arc from Ti to Tj if the two transaction conflict, and Ti accessed the data item before Tj

| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| | | R(B) |
| R(A) | | |
| | W(B) | |
| | | R(A) |
| | W(A) | |

Original Schedule S




Precedence Graph – depicting Conflict pairs

| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| R(A) | | |
| | | R(B) |
| | | R(A) |
| | W(B) | |
| | W(A) | |

Serial Schedule

**Result of Precedence Graph :**

**acyclic - the schedule is conflict serializable**

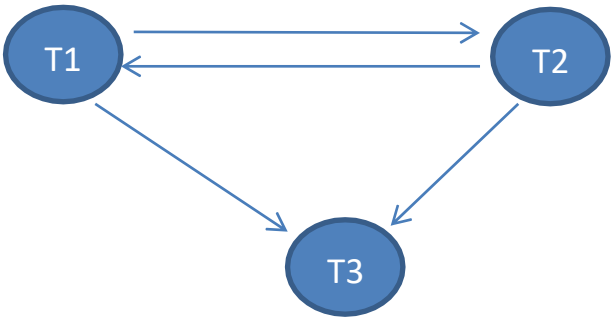
**cyclic - the schedule is not conflict serializable**

If there is no cycle in the precedence graph, it means we can construct a serial schedule S' which is conflict equivalent to schedule S

In the above example, there is no cycle in the graph, means we can construct a serial schedule by swapping non-conflicting actions creating a conflict equivalent schedule.

**Example – Non Conflict Serializable Schedules**

| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| R(B) | | |
| | W(A) | |
| W(A) | | |
| | | W(A) |




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


Note: Both the precedence graphs result in a loop (cycle) Hence the schedules are not conflict serializable.

# Serializability Part II

## View Serializability:

Consider two schedules S and S' , where the same set of transactions participates in both schedules. The schedules S and S 'are said to be view equivalent if three conditions are met:

1. **Initial Read** For each data item Q, if transaction Ti reads the initial value of Q in schedule S, then transaction Ti must, in schedule S , also read the initial value of Q.

2. **Update** For each data item Q, if transaction Ti executes read(Q) in schedule S, and if that value was produced by a write(Q) operation executed by transaction Tj, then the read(Q) operation of transaction Ti must, in schedule S value of Q that was produced by the same write(Q) operation of transaction Tj.

3. **Final Write:** For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q)operation in schedule S'.

Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.

In our previous examples, schedule 1 is not view equivalent to schedule 2, since, in schedule 1, the value of account A read by transaction T2 was produced by T1, whereas this case does not hold in schedule 2. However, schedule 1 is view equivalent to schedule 3, because the values of account A and B read by transaction T2 were produced by T1 in both schedules.

The concept of view equivalence leads to the concept of view serializability. We say that a schedule S is view serializable if it is view equivalent to a serial schedule. As an illustration, suppose that we augment schedule 7 with transaction T6,and obtain schedule 9 in Figure below:

| $T_3$ | $T_4$ | $T_6$ |
|---|---|---|
| read($Q$) | | |
| | write($Q$) | |
| write($Q$) | | |
| | | write($Q$) |

2 Schedule 9—a view-serializable schedule.

Schedule 9 is view serializable. Indeed, it is view equivalent to the serial schedule <T3,T4,T6>, since the one read(Q) instruction reads the initial value of Q in both schedules, and T6 performs the final write of Q in both schedules.

Every conflict-serializable schedule is also view serializable, but there are view serializable schedules that are not conflict serializable. Indeed, schedule 9 is not conflict serializable, since every pair of consecutive instructions conflicts, and, thus, no swapping of instructions is possible.

Observe that, in schedule 9, transactions T4 and T6 perform write(Q)operation without having performed a read(Q) operation. Writes of this sort are called blind writes. Blind writes appear in any view-serializable schedule that is not conflict serializable.

**Example of Schedule that is View Serializable**

| S1 | |
|---|---|
| **T1** | **T2** |
| | **R(A)** |
| **W(A)** | |
| | **W(A)** |
| **R(B)** | |
| **W(B)** | |

| S2 | |
|---|---|
| **T1** | **T2** |
| **W(A)** | |
| | **R(A)** |
| | **W(A)** |
| **R(B)** | |
| **W(B)** | |

- Both S1 and S2 have the same initial read of A by T2
- Both S1 and S2 have the final write of A by T2
- For intermediate writes/reads,
  in S2, T2 reads the value of A after T1 has written to it.
  in S1, T2 reads A which can be viewed as if it read the value after T1 (even though in actual sequence T2 read it before T1 wrote it). The important aspect is the view or effect is equivalent.
- B is read and then written by T1 in both schedules.

Considering the above conditions, S1 and S2 are view equivalent. Thus, if S1 is serializable, S2 is also view serializable.

## Example 2:

Unlike conflict serializability, which cares about the order of conflicting operations, view serializability only cares about the final outcome.

Consider the given example, of two view serializable schedules, which results in the same final output.

S1

| Transaction T1 | Transaction T2 |
|---|---|
| R(X) | |
| W(X) | |
| | R(X) |
| | W(X) |
| R(Y) | |
| W(Y) | |
| | R(Y) |
| | W(Y) |

S2

| Transaction T1 | Transaction T2 |
|---|---|
| R(X) | |
| W(X) | |
| R(Y) | |
| W(Y) | |
| | R(X) |
| | W(X) |
| | R(Y) |
| | W(Y) |

# Recoverability

Recoverability refers to the ability of a system to restore its state to a point where the integrity of its data is not compromised, especially after a failure or an error.

When multiple transactions are executing concurrently, issues may arise that affect the system's recoverability. The interaction between transactions, if not managed correctly, can result in scenarios where a transaction's effects cannot be undone, which would violate the system's integrity.

## Importance of Recoverability:

The need for recoverability arises because databases are designed to ensure data reliability and consistency. If a system isn't recoverable:

- The integrity of the data might be compromised.
- Business processes can be adversely affected due to corrupted or inconsistent data.
- The trust of end-users or businesses relying on the database will be diminished.

## Levels of Recoverability:

a

There are several levels of recoverability, each providing different guarantees about the state of the database after a crash or other failure.

## Recoverable Schedules

A schedule is said to be recoverable if, for any pair of transactions Ti and Tj, if Tj reads a data item previously written by Ti, then Ti must commit before Tj commits. If a transaction fails for any reason and needs to be rolled back, the system can recover without having to rollback other transactions that have read or used data written by the failed transaction.

Suppose we have two transactions T1 and T2.

| T1 | T2 |
|---|---|
| W(A) | |
| | R(A) |
| COMMIT | |
| | W(A) |
| | COMMIT |

In the above schedule, T2 reads a value written by T1, but T1 commits before T2, making the schedule recoverable.

Recoverable schedules avoid the problem of cascading aborts, where the failure of one transaction requires the rollback of multiple other transactions.Example of a Recoverable Schedule

## Non-Recoverable Schedules

A schedule is said to be non-recoverable (or irrecoverable) if there exists a pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, but Ti has not committed yet and Tj commits before Ti. If Ti fails and needs to be rolled back after Tj has committed, there's no straightforward way to roll back the effects of Tj, leading to potential data inconsistency.

**Example of a Non-Recoverable Schedule**

Again, consider two transactions T1 and T2.

| T1 | T2 |
|---|---|
| W(A) | |
| | R(A) |
| | W(A) |
| | COMMIT |
| COMMIT | |

In this schedule, T2 reads a value written by T1 and commits before T1 does. If T1 encounters a failure and has to be rolled back after T2 has committed, we're left in a problematic situation since we cannot easily roll back T2, making the schedule non-recoverable.

This is generally undesirable and is avoided in most systems.

**Cascading Rollback**

A cascading rollback occurs when the rollback of a single transaction causes one or more dependent transactions to be rolled back. This situation can arise when one transaction reads uncommitted changes of another transaction, and then the latter transaction fails and needs to be rolled back. Consequently, any transaction that has read the uncommitted changes of the failed transaction also needs to be rolled back, leading to a cascade effect.

**Example of Cascading Rollback**

Consider two transactions T1 and T2:

| T1 | T2 |
|---|---|
| W(A) | |
| | R(A) |
| | R(A) |
| ABORT | |
| ROLLBACK | |
| | ROLLBACK |

Here, T2 reads an uncommitted value of A written by T1. When T1 fails and is rolled back, T2 also has to be rolled back, leading to a cascading rollback. This is undesirable because it wastes computational effort and can complicate recovery procedures.

## Cascadeless Schedules

A schedule is considered cascadeless if transactions only read committed values. This means, in such a schedule, a transaction can read a value written by another transaction only after the latter has committed. Cascadeless schedules prevent cascading rollbacks.

**Example of Cascadeless Schedule**

Consider two transactions T1 and T2:

| T1 | T2 |
|---|---|
| W(A) | |
| COMMIT | |
| | R(A) |
| | W(B) |
| | COMMIT |

In this schedule, T2 reads the value of A only after T1 has committed. Thus, even if T1 were to fail before committing (not shown in this schedule), it would not affect T2. This means there's no risk of cascading rollback in this schedule.

While this improves consistency and reduces the complexity of recovery, it may limit concurrency, as transactions must wait for others to commit before accessing the data.

# DATABASE MANAGEMENT SYSTEMS

## Implementation of Isolation

Isolation is one of the core ACID properties of a database transaction, ensuring that the operations of one transaction remain hidden from other transactions until completion. It means that no two transactions should interfere with each other and affect the other's intermediate state.

## Isolation Levels

Isolation levels defines the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system.

There are four levels of transaction isolation defined by SQL -

## 1. Serializable

- This is the highest level of isolation, ensuring complete isolation from other transactions. It appears as if transactions are executed one after another, serially.
- Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

## 2. Repeatable Read

- This is the most restrictive isolation level.
- The transaction holds read locks on all rows it references.
- It holds write locks on all rows it inserts, updates, or deletes.
- Since other transaction cannot read, update or delete these rows, it avoids non repeatable read.

## 3. Read Committed

A transaction can only see changes committed before it started; i.e. the any data is committed immediately after its access.

- This isolation level allows only committed data to be read.
- Thus it does not allows dirty read (i.e. one transaction reading of data immediately after written by another transaction).
- The transaction hold a read or write lock on the current row, and thus prevent other rows from reading, updating or deleting it.

## 4. Read Uncommitted

Transactions may see uncommitted changes made by other transactions.

- It is lowest isolation level.
- In this level, one transaction may read not yet committed changes made by other transaction.
- This level allows dirty reads.

The choice of isolation level depends on the specific requirements of the application. Higher isolation levels offer stronger data consistency, but can result in longer lock times and increased contention, leading to decreased concurrency and performance.

Lower isolation levels provide more concurrency, but can result in data inconsistencies**.**

## Implementation of Isolation

Implementing isolation typically involves concurrency control mechanisms. Here are common mechanisms used:

## 1. Locking Mechanisms

Locking ensures exclusive access to a data item for a transaction. This means that while one transaction holds a lock on a data item, no other transaction can access that item.

## 2. Timestamp-based Protocols

Every transaction is assigned a unique timestamp when it starts. This timestamp determines the order of transactions. Transactions can only access the database if they respect the timestamp order, ensuring older transactions get priority.

## Lock Based Protocols:

In lock-based protocol, a transaction cannot access or write data unless it obtains a suitable lock. The Two-Phase Locking Protocol ensures transaction consistency by employing a set of rules. Transactions must obtain appropriate locks before accessing or modifying data, preventing conflicts and maintaining data integrity throughout the transaction's lifecycle.

**Locks are essential in a database system to ensure:**

**Consistency:** Without locks, multiple transactions could modify the same data item simultaneously, resulting in an inconsistent state.

**Isolation:** Locks ensure that the operations of one transaction are isolated from other transactions, i.e., they are invisible to other transactions until the transaction is committed.

**Concurrency:** While ensuring consistency and isolation, locks also allow multiple transactions to be processed simultaneously by the system, optimizing system throughput and overall performance.

**Avoiding Conflicts:** Locks help in avoiding data conflicts that might arise due to simultaneous read and write operations by different transactions on the same data item.

**Preventing Dirty Reads**: With the help of locks, a transaction is prevented from reading data that hasn't yet been committed by another transaction.

### Types of Locks Used in Transaction Control

In transaction control, there are two common types of locks shared lock and exclusive lock. Here is a brief overview about it:

**1. Shared lock**: The transactions can only read the data items in a shared lock.

While a transaction attempting to update the data will be unable to do so until the shared lock is freed, a transaction attempting to read the same data will be allowed to do so.

Read lock is another name for shared lock, which is exclusively used for reading data objects.

**2. Exclusive lock**: The transactions can read and write to the data items in an exclusive lock.

When a statement updates data, its transaction has an exclusive lock on the modified data, preventing access by other transactions, until the transaction holding the lock commits or rolls it back, the lock is in effect.

## The difference between shared lock and exclusive lock

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

# DATABASE MANAGEMENT SYSTEMS

## Lock Based Protocols

**What is Two-phase locking (2PL)?**

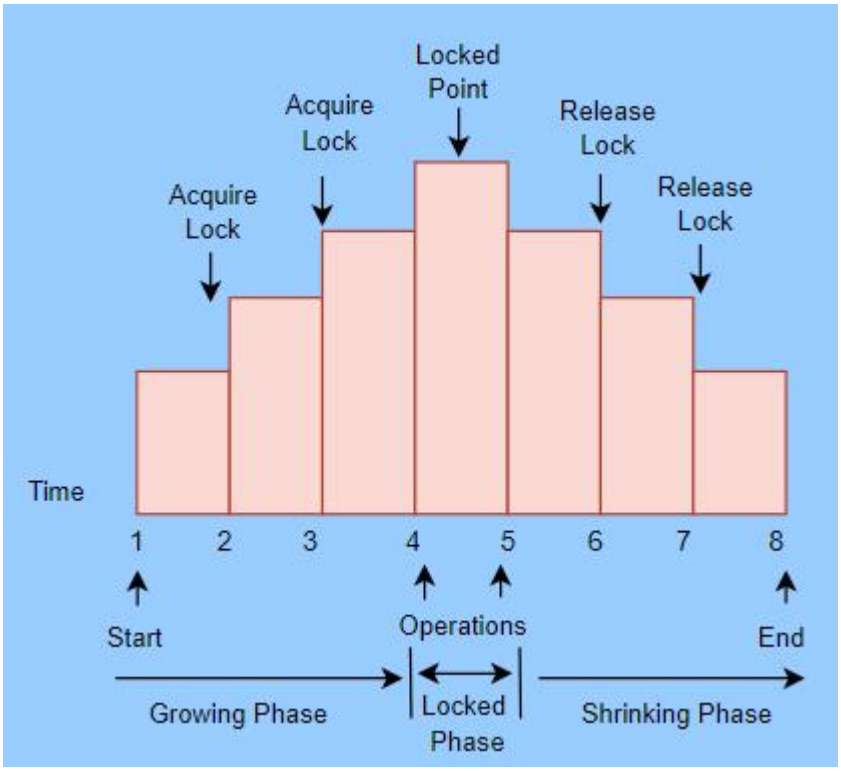
The two-phase locking divides the transaction execution phase into three components.

- When the transaction's execution begins in the first part, it requests permission for the lock it requires.
- The transaction then obtains all of the locks in the second part. The third phase begins when the transaction's first lock is released.
- The transaction cannot demand any new locks in the third phase. It only unlocks the locks that have been acquired.

A transaction follows the two-phase locking protocol if locking and unlocking can be done in two phases.

**The two phases of the two-phase protocol are:**

- **Growing phase:** During the growth phase, new locks on data items may be acquired, but none can be released.
- **Shrinking phase:** Existing locks may be released, but no new locks can be acquired during the shrinking phase.




The depicted diagram illustrates the growing phase, where locks are acquired until all necessary transaction locks are obtained, defining the Lock-Point. Beyond this point, transactions transition into the Shrinking phase, ensuring a structured lock-based protocol.

**Lock Point**
It is a point at which the growing phase comes to a close, i.e., when a transaction obtains the last lock it requires.

Let's look at an example:

| | T1 | T2 |
|---|---|---|
| 1 | Lock - S(A) | |
| 2 | | Lock - S(A) |
| 3 | Lock - X(B) | |
| 4 | ……. | ……. |
| 5 | Unlock(A) | |
| 6 | | Lock - X(C) |
| 7 | Unlock(B) | |
| 8 | | Unlock(A) |
| 9 | | Unlock(C) |
| 10 | ……. | ……. |

The way unlocking and locking works in two-phase locking is:

**Transaction T1:**
- The growing phase is from steps 1-3.
- The shrinking phase is from steps 5-7.
- The locking point is at point 3.

**Transaction T2:**
- The growing phase is from steps 2-6.
- The shrinking phase is from steps 8-9.
- The locking point is at point 6.

The type mentioned above of 2-PL is Basic 2PL. It ensures Serializability, but it does not prevent Cascading Rollback and Deadlock.

## Advantages and Disadvantages of Two-Phase Locking Protocol

## Pros of Two-Phase Locking (2PL)

**Ensures Serializability:** 2PL guarantees conflict-serializability, ensuring the consistency of the database.

**Concurrency:** By allowing multiple transactions to acquire locks and release them, 2PL increases the concurrency level, leading to better system throughput and overall performance.

**Avoids Cascading Rollbacks**: Since a transaction cannot read a value modified by another uncommitted transaction, cascading rollbacks are avoided, making recovery simpler.

## Cons of Two-Phase Locking (2PL)

**Deadlocks:** Two or more transactions wait indefinitely for a resource locked by the other.

**Reduced Concurrency** (in certain cases): If one transaction holds a lock for a long time, other transactions needing that lock will be blocked.

**Overhead**: There's a time cost associated with acquiring and releasing locks, and memory overhead for maintaining the lock table.

**Starvation:** It's possible for some transactions to get repeatedly delayed if other transactions are continually requesting and acquiring locks

## Types of Two-Phase Locking Protocol
## Two Phase Locking is classified into three types :

## 1. Strict two-phase locking protocol

- Exclusive(X) locks held by the transaction be released after committing in addition to the lock being 2-Phase.
- Ensures that the schedule is recoverable and cascadeless.

**2. Rigorous two-phase locking protocol**

- Exclusive(X) and Shared(S) locks held by the transaction be released after Transaction Commits, in addition to the lock being 2-Phase. This means there is no explicit shrinking phase where locks are released during the transaction. Instead, locks are released only at the end of the transaction.

- It ensures that schedule is recoverable and cascadeless.

**3. Conservative two-phase locking protocol**

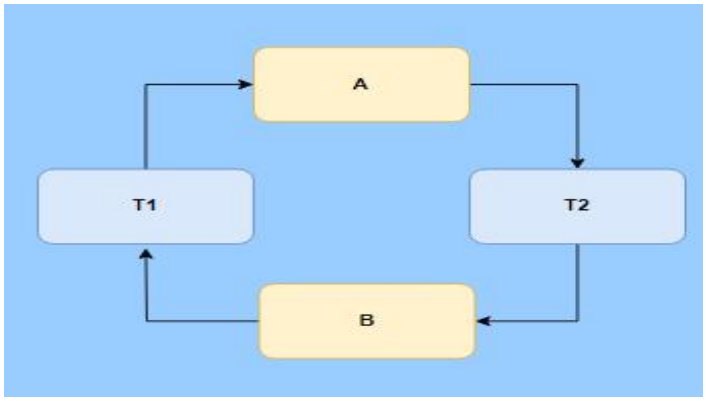
- Static 2-PL enforces pre-locking of all accessed resources by declaring read-set and write-set before transaction execution.
- Transaction waits if necessary items can't be locked, avoiding locking any items if a predeclared requirement isn't met.

**Deadlock in a two-phase locking**

Deadlocks are possible in a 2PL. You can detect the deadlocks by drawing the precedence graph of the transactions schedule, and if you find a loop, then there is a deadlock situation.

Deadlocks are needed to be resolved to complete the transaction. A deadlock is resolved by aborting a transaction of the given loop and breaking it.

**Let's understand deadlock in a two-phase locking protocol with the help of an example.** Suppose there are two transactions, T1 and T2. If the completion of transaction T1 is dependent on the completion of transaction T2 and the completion of transaction, T2 is dependent on the completion of transaction T1. This results in a state of deadlock. As shown in the figure below:




**Phase 1 - Lock Request:**

- Process T1 acquires a lock on Resource A.
- Process T2 acquires a lock on Resource B.

**Phase 2 - Lock Release:**

- Process T1 now needs Resource B to proceed further, but it cannot release Resource A until it has both resources due to the two-phase locking rule.
- Process T2 similarly needs Resource A to continue, but it's holding onto Resource B.

# DATABASE MANAGEMENT SYSTEMS

## Timestamp Based Protocols

Timestamp-based protocols in database management systems (DBMS) are concurrency control mechanisms used to ensure the consistency of the database while allowing concurrent transactions.

They use timestamps to order the operations of transactions in such a way that the database remains in a consistent state, following a predetermined serialization order. Here are the key concepts and protocols related to timestamp-based concurrency control:

Each transaction is issued a timestamp (can be the system's clock time or a logical counter that increments with each new transaction) when it enters the system. If an old transaction Ti has time-stamp TS(Ti), a new transaction Tj is assigned time-stamp TS(Tj) such that **TS(Ti) <TS(Tj).**

Schedules managed using timestamp based protocol are serializable just like the two phase protocols.

Since older transactions are given priority which means no transaction has to wait for longer period of time that makes this protocol free from deadlock.

### Key Concepts

1. **Timestamp**: A unique identifier assigned to each transaction. It usually represents the time at which the transaction started.
    - **Transaction Timestamp (TS)**: Denoted as TS(Ti) for transaction Ti.
    - **Read Timestamp (RTS)**: Denoted as RTS(X) for a data item X, indicating the largest timestamp of any transaction that successfully read X.
    - **Write Timestamp (WTS)**: Denoted as WTS(X) for a data item X, indicating the largest timestamp of any transaction that successfully wrote X.

### Basic Timestamp Ordering Protocol

The basic idea is to ensure that transactions are executed in a serial order consistent with their timestamps. For this:

1. **Read Operation**:
    - Transaction TiT_iTi requests to read a data item X.
    - If TS(Ti)<WTS(X), it means a younger transaction has already written X, so Ti is rolled back.
    - If TS(Ti)≥WTS(X), the read is allowed, and RTS(X) is updated to TS(Ti), if TS(Ti)>RTS(X).

2. **Write Operation**:
    - Transaction Ti requests to write a data item X.
    - If TS(Ti)<RTS(X), it means a younger transaction has already read X, so Ti is rolled back.
    - If TS(Ti)<WTS(X), it means a younger transaction has already written X, so Ti is rolled back.
    - If TS(Ti)≥RTS(X) and TS(Ti)≥WTS(X), the write is allowed, and WTS(X) is updated to TS(Ti)

**Advantages**

- **Serializability**: Ensures serializable schedules, maintaining database consistency.
- **Deadlock-free**: Avoids deadlocks since transactions are never made to wait; they are either allowed to proceed or rolled back.

**Disadvantages**

- **Cascading Rollbacks**: A rollback of a transaction may cause other dependent transactions to roll back.
- **Long Transactions**: Long-running transactions may repeatedly get rolled back if newer transactions conflict with them.
- **Storage Overhead**: Requires additional storage for timestamps and may involve maintaining multiple versions of data items.

**Implementation Example**

Let's consider an example to illustrate how a basic timestamp ordering protocol works:

**Note:** The initial values of the Write Timestamp (WTS) and Read Timestamp (RTS) for a data item X are typically set to zero or negative infinity ($-\infty$) before any transactions have accessed the data item. This initialization ensures that the first read or write operation on X by any transaction will be allowed.

- Suppose there are two transactions T1 and T2 with timestamps TS(T1) = 1 and TS(T2) = 2. Let RTS(X) =0 and WTS(X)=0.
-
- **T1 reads X:**
    - Since TS(T1)≥WTS(X), T1 is allowed to read X.
    - Update RTS(X) to 1.
    -
- **T2 writes X:**
    - Since TS(T2)≥RTS(X) and TS(T2)≥WTS(X) T2 is allowed to write X.
    - Update WTS(X) 2.
- **T1 writes X:**
    - Since TS(T1)<WTS(X) T1 is rolled back because a newer transaction T2 has already written X.

In this way, timestamp-based protocols manage the order of transaction operations to maintain consistency while allowing concurrent execution.

## Thomas' Write Rule:

Thomas' Write Rule is an optimization in timestamp-based concurrency control protocols. It allows certain write operations to be ignored instead of causing the transaction to roll back, thereby improving the system's performance and reducing unnecessary rollbacks.

**If a transaction Ti attempts to write a data item X and TS(Ti)<WTS(X), instead of rolling back Ti, the write operation is simply ignored.**

**Example Scenario:**

Let's illustrate Thomas' Write Rule with the above example involving two transactions T1 and T2.

**The second attempt T1 wants to Write X after T2 has written:**

- T1 requests to write X.
- The basic timestamp ordering protocol would check TS(T1)<WTS(X). Here, TS(T1)=1 and WTS(X)=2. Since TS(T1)<WTS(X), the basic protocol would roll back T1.
- With Thomas' Write Rule, instead of rolling back T1, the write operation is ignored because the write by T1 is considered obsolete due to the later write by T2.

**Why Thomas' Write Rule Works:**

Thomas' Write Rule optimizes concurrency control by reducing unnecessary rollbacks:

- **Efficiency**: By ignoring obsolete writes, the system avoids the overhead of rolling back transactions that do not affect the final state of the database.
- **Maintains Serializability**: It ensures the serializability of transactions by only allowing the most recent write to a data item to take effect.

# DATABASE MANAGEMENT SYSTEMS
# Validation Based Protocols

Validation-based protocols, also known as Optimistic Concurrency Control (OCC), are a set of techniques that aim to increase system concurrency and performance by assuming that conflicts between transactions will be rare.

Instead of preventing conflicts through locking, these protocols validate transactions at commit time to ensure that no conflicts have occurred.

In optimistic concurrency control, the basic premise is that most transactions will not conflict with other transactions, and the idea is to be as permissive as possible in allowing transactions to execute.

## The Three Phases:

### 1. Read Phase

- Transactions execute and read data items into a local workspace.
- No updates are made to the database during this phase.

### 2. Validation Phase

- At the end of the read phase, the transaction enters the validation phase.
- The system checks whether committing the transaction would violate serializability.
- Validation involves comparing the transaction's read set and write set against those of other transactions.

### 3. Write Phase

- If the transaction passes validation, it proceeds to the write phase.
- The updates from the local workspace are applied to the database.
- If the transaction fails validation, it is rolled back and may be restarted.

## Validation Criteria

### Timestamp Ordering

- Each transaction is assigned a timestamp when it starts.
- The validation ensures that the transaction's operations can be serialized in the order of their timestamps.

## Validation Rules

**Rule 1:** For two transactions Ti and Tj, where Ti completes its read phase before Tj starts, no validation is needed since they do not overlap.

**Rule 2:** If Ti overlaps with Tj and Ti precedes Tj in the validation order, Tj's read set must not intersect with Ti's write set.

**Rule 3:** If Ti and Tj overlap and Tj precedes Ti in the validation order, Ti's write set must not intersect with Tj's read set and write set.

## Example Scenarios

### Example 1: Non-Conflicting Transactions

- **T1** reads data items A and B, then writes to B.
- **T2** reads data items C and D, then writes to D.
- Since T1 and T2 operate on different data items, they will both pass validation and commit successfully.

### Example 2: Conflicting Transactions

- **T1** reads data items A and B, then writes to B.
- **T2** reads data items B and C, then writes to C.
- During validation, if T2's read set overlaps with T1's write set (both accessing B), validation may fail depending on the order and timing of the transactions.

## Advantages and Disadvantages

### Advantages

- **Increased Concurrency**: Transactions proceed without waiting for locks, increasing parallelism.
- **Reduced Deadlocks**: Since locks are not used extensively, the risk of deadlocks is minimized.
- **Simplicity**: The protocol is simpler to implement for read-mostly workloads where conflicts are rare.

### Disadvantagess

- **High Overhead**: Rolling back and restarting transactions that fail validation can be costly.
- **Resource Intensive**: Maintaining read and write sets and performing validation checks require additional resources.
- **Not Suitable for High-Conflict Workloads**: In environments with frequent conflicts, the overhead of rollbacks can outweigh the benefits.

# DATABASE MANAGEMENT SYSTEMS
## Multiple Granularity

A lock guarantees exclusive use of a data item to a current transaction. A transaction acquires a lock prior to data access; the lock is released when the transaction is completed so that another transaction can lock the data item for its exclusive use. The use of locks based on the assumption that conflict between transactions is likely to occur is often referred to as pessimistic locking. During a transaction; the database might be in a temporary inconsistent state when several updates are executed. Therefore, locks are required to prevent another transaction from reading inconsistent data..

Most multiuser DBMSs automatically initiate and enforce locking procedures. All lock information is managed by a lock manager, which is responsible for assigning and policing the locks used by the transactions.

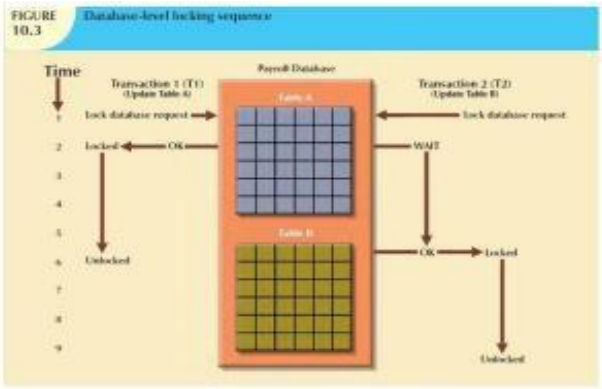
## Lock Granularity:

Lock granularity indicates the level of lock use. Multiple granularity locking manages locks at various levels of a database hierarchy, allowing different transactions to lock resources at different granular levels, such as the entire database, tables, pages, or individual records or even fields.

## Granularity Levels

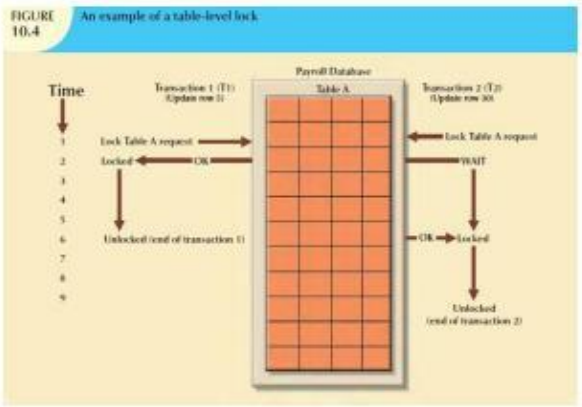
**Database Level: The highest level of granularity, locking the entire database.**

In a database-level lock, the entire database is locked, thus preventing the use of any tables in the database by transaction T2 while transaction Tl is being executed. This level of locking is good for batch processes, but it is unsuitable for multiuser DBMSs. Note that because of the database-level lock, transactions T1 and T2 cannot access the same database concurrently even when they use different tables.



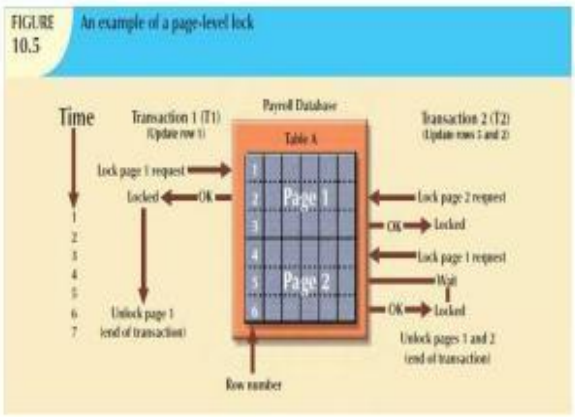

**File/Table Level: Locks an entire file or table.**

In a table-level lock, the entire table is locked, preventing access to any row by transaction T2 while transaction T1 is using the table. If a transaction requires access to several tables, each table may be locked. However, two transactions can access the same database as long as they access different tables. Table-level locks, while less restrictive than database-level locks, cause traffic jams when many transactions are waiting to access the same table.



Consequently, table-level locks are not suitable for multiuser DBMSs. Note that in Figure, transactions T1 and T2 cannot access the same table even when they try to use different rows; T2 must wait until T1 unlocks the table.

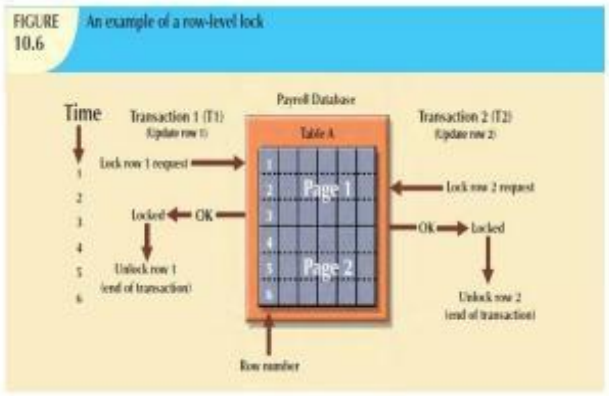
**Page Level: Locks a specific page within a file or table.**




In a page-level lock, the DBMS will lock an entire diskpage. A diskpage, or page, is the equivalent of a diskblock, which can be described as a directly addressable section of a disk. A page has a fixed size, such as 4K, 8K, or 16K. A table can span several pages, and a page can contain several rows of one or more tables. Page-level locks are currently the most frequently used multiuser DBMS locking method.

An example of a page-level lock is shown in the Figure. Note that T1 and T2 access the same table while locking different diskpages. If T2 requires the use of a row located on a page that is locked by T1, T2 must wait until the page is unlocked by T1.

**Record Level: Locks a specific record within a page.**




A row-level lock is much less restrictive than the locks discussed earlier. The DBMS allows concurrent transactions to access different rows of the same table even when the rows are located on the same page. Although the row-level locking approach improves the availability of data, its management requires high overhead because a lock exists for each row in a table of the database involved in a conflicting transaction.

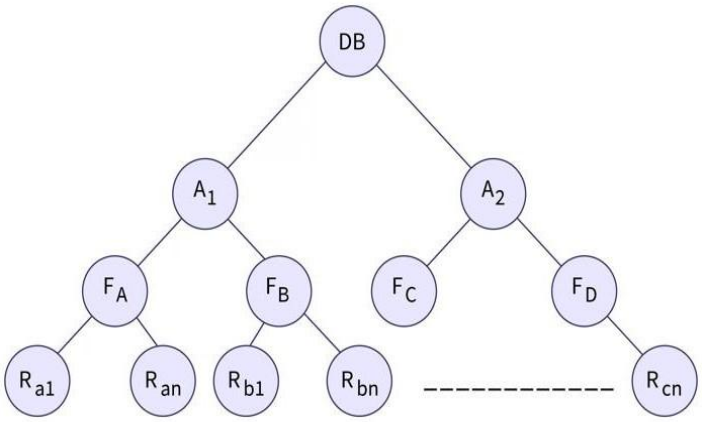
**Field Level: The most granular level, locking specific fields within a record.**

The field-level lock allows concurrent transactions to access the same row as long as they require the use of different fields (attributes) within that row. Although field-level locking clearly yields the most flexible multiuser data access, it is rarely implemented in a DBMS because it requires an extremely high level of computer overhead and because the row-level lock is much more useful in practice.

## Hierarchical Locking Structure

Locks are organized in a hierarchical tree structure, with each node representing a different level of granularity.

Parent-child relationships ensure that locks acquired at a higher level (parent) must be compatible with locks at lower levels (child).




In addition to shared (S) and exclusive (X) locks, multiple-granularity locking protocols also use two new kinds of locks, called **"intention locks**." These locks indicate a transaction's intention to acquire a finer-grained lock in the future.

## Types of Intention Locks

**Intention Shared (IS):** Indicates intention to acquire shared locks at a lower level. When a Transaction needs S lock on a node "K", the transaction would need to apply IS lock on all the precedent nodes of "K", starting from the root node. So, when a node is found locked in IS mode, it indicates that some of its descendent nodes must be locked in S mode.

**Example:** Suppose a transaction wants to read a few records from a table but not the whole table. It might set an IS lock on the table, and then set individual S locks on the specific rows it reads.

**Intention Exclusive (IX):** Indicates intention to acquire exclusive locks at a lower level. When a Transaction needs X lock on a node "K", the transation would need apply IX lock on all the precedent nodes of "K", starting from the root node. So, when a node is found locked in IX mode, it indicates that some of its descendent nodes must be locked in X mode.

**Example:** If a transaction aims to update certain records within a table, it may set an IX lock on the table and subsequently set X locks on specific rows it updates.

**Shared Intention Exclusive (SIX):** A combination of shared lock at the current level and intention to acquire exclusive locks at a lower level. When a node is locked in SIX mode; it indicates that the node is explicitly locked in S mode and Ix mode. So, the entire tree rooted by that node is locked in S mode and some nodes in that are locked in X mode. This mode is compatible only with IS mode.

**Example:** Suppose a transaction wants to read an entire table but also update certain rows. It would set a SIX lock on the table. This tells other transactions they can read the table but cannot update it until the SIX lock is released. Meanwhile, the original transaction can set X locks on specific rows it wishes to update.

## Compatibility Matrix with Lock Modes in multiple granularity

A compatibility matrix defines which types of locks can be held simultaneously on a database object. Here's a simplified matrix:

| | IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| **IS** | Y | Y | Y | N | N |
| **IX** | Y | Y | N | N | N |
| **S** | Y | N | Y | N | N |
| **SIX** | N | N | N | N | N |
| **X** | N | N | N | N | N |

## The Scheme operates as follows:-

- A Transaction must first lock the Root Node and it can be locked in any mode.
- Locks are granted as per the Compatibility Matrix indicated above.
- A Transaction can lock a node in S or IS mode if it has already locked all the predecessor nodes in IS or IX mode.
- A Transaction can lock a node in X or IX or SIX mode if it has already locked all the predecessor nodes in SIX or IX mode.
- A transaction must follow two-phase locking. It can lock a node, only if it has not previously unlocked a node. Thus, schedules will always be conflict-serializable.
- Before it unlocks a node, a Transaction has to first unlock all the children nodes of that node**. Thus, locking will proceed in top-down manner and unlocking will proceed in bottom-up manner. This will ensure the resulting schedules to be deadlock-free.**

**Example**

1. Transaction T1 wants to read a record R in table T:
    - Acquire IS lock on database.
    - Acquire IS lock on table T.
    - Acquire S lock on record R.

2. Transaction T1 and T2 operate on the same table but different records:
    - T1 acquires
        - IS lock on database,
        - IS lock on table, and
        - S lock on record R1.
    - T2 can concurrently acquire
        - IS lock on database,
        - IS lock on table, and
        - S lock on record R2.

## Advantages and Disadvantages

## Advantages

- Improved Concurrency: Allows multiple transactions to operate concurrently on different parts of the database.
- Reduced Lock Contention: By allowing more granular locking, it reduces the chance of lock contention.
- Flexibility: Provides flexibility in managing locks at various levels of granularity.

## Disadvantages

- Complexity: More complex to implement and manage compared to single-level locking.
- Overhead: Increased overhead due to maintaining and checking multiple levels of locks

# DATABASE MANAGEMENT SYSTEMS
# Recovery and Atomicity Part - I

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items.

Database recovery means the process of restoring the database to a correct state in case of failures.

## Failures can be of different types, such as:

**Transaction Failure:** When a transaction cannot compete successfully due to logical errors, system errors, or data errors.

**System Crash:** When the DBMS crashes due to hardware failures, software bugs, or power outages.

**Media Failure:** When the storage media (like hard disks) fail, leading to loss of data.

**When a DBMS recovers from a crash, it should maintain the following −**

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

## Types of techniques for recovery:

**Maintaining logs of each transaction** - In case of a failure, the system can use these logs to redo or undo transactions to restore the database to a consistent state.

**Checkpointing:** Periodically saves the state of the database. During a failure, the system can roll back to the last checkpoint and then apply the logs to reach a consistent state.

**Maintaining shadow paging** - If a failure occurs, the system can revert to the shadow copy.

## Log Based Recovery:

Log-based recovery is a widely used approach in database management systems to recover from system failures and maintain atomicity and durability of transactions.

The fundamental idea behind log-based recovery is to keep a log of all changes made to the database, so that after a failure, the system can use the log to restore the database to a consistent state.

## Transaction Log

A log in DBMS is a crucial component used for ensuring data integrity, consistency, and facilitating recovery processes.

A log, also known as a **transaction log or write-ahead log (WAL),** is a sequential record that tracks all the changes made to the database. It records information about transactions before they are committed to the database. This information can be used to redo completed transactions or undo incomplete transactions in the event of a failure.

A log typically contains the following information:

- Transaction ID: A unique identifier for each transaction.
- Transaction Type: The type of operation, such as INSERT, UPDATE, DELETE, etc.
- Affected Data: Details of the data being modified, - table name and the specific rows and columns.
- Beoref-Image: The state of the data before the transaction (used for undo operations).
- After-Image: The state of the data after the transaction (used for redo operations).
- Timestamp: The time at which the transaction was recorded in the log.
- Log Sequence Number (LSN): A unique identifier for each log entry, ensuring the sequence of transactions can be tracked.

For every transaction that modifies the database, an entry is made in the log. This entry typically includes:

- Transaction ID: A unique identifier for the transaction.
- Data item identifier: Identifier for the specific item being modified.
- OLD value: The value of the data item before the modification.
- NEW value: The value of the data item after the modification.

We represent an update log record as **<Ti , Xj , V1, V2>,** - transaction Ti has performed a write on data item Xj. Xj had value V1 before the write, and has value V2 after the write.

Some other types of log records are:

**<Ti start>.** Transaction Ti has started.

**<Ti commit>**. Transaction Ti has committed.

**<Ti abort>.** Transaction Ti has aborted.

## How Log-Based Recovery works

### 1. Transaction Logging:

For every transaction that modifies the database, an entry is made in the log.

### 2. Writing to the Log

Before any change is written to the actual database (on disk), the corresponding log entry is stored. This is called the Write-Ahead Logging (WAL) principle. By ensuring that the log is written first, the system can later recover and apply or undo any changes.

**3. Checkpointing**

A checkpoint is a point of synchronization between the database and its log. At the time of a checkpoint:

All the changes in main memory (buffer) up to that point are written to disk.

A special entry is made in the log indicating a checkpoint. This helps in reducing the amount of log that needs to be scanned during recovery.

**4. Recovery Process**

Redo: If a transaction is identified (from the log) as having committed but its changes have not been reflected in the database (due to a crash before the changes could be written to disk), then the changes are reapplied using the 'After Image' from the log.

Undo: If a transaction is identified as not having committed at the time of the crash, any changes it made are reversed using the 'Before Image' in the log to ensure atomicity.

**5. Commit/Rollback**

Once a transaction is fully complete, a commit record is written to the log. If a transaction is aborted, a rollback record is written, and using the log, the system undoes any changes made by this transaction.

## Deferred-Write Technique

Transaction recovery procedures generally make use of deferred-write and write-through techniques.

**Deferred-write technique (also called a deferred update),**

The transaction operations do not immediately update the physical database. Instead, only the transaction log is updated. The database is physically updated only after the transaction reaches its commit point, using information from the transaction log.

The recovery process for all started and committed transactions (before the failure) follows these steps:

1. Identify the last checkpoint in the transaction log. This is the last time transaction data was physically saved to disk.

2. For a transaction that started and was committed before the last checkpoint, nothing needs to be done because the data are already saved.

3. For a transaction that performed a commit operation after the last checkpoint, the DBMS uses the transaction log records to redo the transaction and to update the database, using the ―after values in the transaction log.

4. For any transaction that had a ROLLBACK operation after the last checkpoint or that was left active (with neither a COMMIT nor a ROLLBACK) before the failure occurred, nothing needs to be done because the database was never updated.

## Write-through Technique

Write-through technique (also called an immediate update),

the database is immediately updated by transaction operations during the transaction‘s execution, even before the transaction reaches its commit point. If the transaction aborts before it reaches its commit point, a ROLLBACK or undo operation needs to be done to restore the database to a consistent state. In that case, the ROLLBACK operation will use the transaction log ―before values.

The recovery process follows these steps:

1. Identify the last checkpoint in the transaction log. This is the last time transaction data were physically saved to disk.

2. For a transaction that started and was committed before the last checkpoint, nothing needs to be done because the data are already saved.

3. For a transaction that was committed after the last checkpoint, the DBMS redoes the transaction, using the ―after values of the transaction log.

4. For any transaction that had a ROLLBACK operation after the last checkpoint or that was left active (with neither a COMMIT nor a ROLLBACK) before the failure occurred, the DBMS uses the transaction log records to ROLLBACK or undo the operations, using the before values in the transaction log.

## Benefits of Log-Based Recovery

- **Atomicity:** Guarantees that even if a system fails in the middle of a transaction, the transaction can be rolled back using the log.

- **Durability:** Ensures that once a transaction is committed, its effects are permanent and can be reconstructed even after a system failure.

- **Efficiency:** Since logging typically involves sequential writes, it is generally faster than random access writes to a database.

# DATABASE MANAGEMENT SYSTEMS
# Recovery and Atomicity Part - II

## Shadow Paging

Shadow paging is a recovery mechanism used in Database Management Systems (DBMS) to ensure data integrity and consistency without requiring extensive logging.
The basic idea is to maintain two versions of the database pages:

current pages - that are being actively modified and the

shadow pages - that represent a stable, consistent state of the database

**Overview of Shadow Paging**

In shadow paging, the database is divided into fixed-size blocks or pages. Two page tables are maintained:

**Current Page Table (CPT):** This table points to the current version of the pages.

**Shadow Page Table (SPT):** This table points to the shadow (or stable) version of the pages.

**Basic Operation:**

1. **Initialization:**
   - At the start, the SPT and CPT are identical, pointing to the same set of pages.
   - The SPT is written to a non-volatile storage medium to ensure it can be recovered after a crash.
2. **Transaction Start:**
   - When a transaction begins, any modification to a page results in the creation of a new version of that page.
   - The original page remains unchanged, preserving the pre-transaction state.
3. **Page Modification:**
   - When a transaction modifies a page, the DBMS creates a copy of that page.
   - The CPT is updated to point to the new version of the page, while the SPT continues to point to the old version
4. **Commit Operation:**
   - When the transaction commits, the CPT is written to the non-volatile storage.
   - At this point, the SPT can be updated to match the CPT, making the new page versions the new stable state.
5. **Rollback Operation:**
   - If a transaction needs to be rolled back, the DBMS simply discards the modified pages and the CPT.
   - The SPT remains unchanged, and the database reverts to its pre-transaction state.

## Advantages of Shadow Paging

- **No Undo Logging Required:** Since the original pages are never overwritten, there is no need to maintain undo logs.
- **Consistency:** The database remains in a consistent state because the shadow pages represent a stable snapshot.
- **Simple Rollback**: Rolling back a transaction is straightforward as it simply involves discarding the current pages and reverting to the shadow.

## Disadvantages of Shadow Paging

- **Copy Overhead:** Every page modification requires copying the page, which can be resource-intensive.
- **Space Utilization:** Maintaining two versions of pages can lead to higher storage requirements.
- **Checkpointing Complexity:** Periodically updating the SPT to match the CPT can be complex and time-consuming, especially for large databases.

# DATABASE MANAGEMENT SYSTEMS
# Recovery with Concurrent Transactions

Concurrency execution allows multiple transactions to execute 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.

Recovery with concurrent transactions can be done in the following four ways.

- Interaction with concurrency control
- Transaction rollback
- Checkpoints
- Restart recovery

## Interaction with concurrency control

Concurrency control mechanisms manage how transactions interact with each other to maintain database consistency. Common techniques include:

- **Locking:** Transactions acquire locks on data items to prevent other transactions from accessing or modifying them concurrently.
- **Timestamp Ordering:** Transactions are ordered based on their timestamps to ensure serializability.
- **Optimistic Concurrency Control:** Transactions proceed without locking, based on the assumption that conflicts are rare. Conflicts are detected and resolved at commit time.

During recovery, these mechanisms play a crucial role in ensuring that transactions are correctly rolled back or redone.

If a failure occurs, the system uses information from the logs and transaction metadata managed by concurrency control to determine which transactions need to be undone (rolled back) or redone.

This ensures that the database returns to a consistent state without violating transaction isolation and consistency guarantees.

<

## Transaction rollback

- Transaction rollback involves undoing the effects of a transaction that has not yet been committed due to an error or abort condition. It restores the database to its state before the transaction began.
- When a system failure occurs, transactions that were in progress or incomplete need to be rolled back during recovery.

- The system uses transaction logs, which record the sequence of operations performed by each transaction, to identify and reverse the changes made by these incomplete transactions.
- This ensures that any uncommitted changes do not persist in the database, maintaining consistency and integrity.

## Checkpointing:

- Checkpoints are periodic snapshots of the database state that are recorded at specific intervals. They capture the committed state of the database at those points in time.
- During recovery, checkpoints serve as reference points to minimize the amount of log data that needs to be processed.
- Instead of replaying all transactions from the beginning of the logs after a crash, the system can start recovery from the most recent checkpoint.
- This optimization reduces the recovery time significantly, as only the transactions that occurred after the last checkpoint need to be reprocessed (redo) or undone (undo).
- Checkpoints thus facilitate faster recovery and reduce the potential for data loss or inconsistency.

## Restart Recovery:

- Restart recovery is the process of bringing the database system back to a consistent state after it has been restarted following a failure. It involves analyzing the transaction logs and applying necessary operations to restore data consistency.
- After a system crash and subsequent restart, the database system initiates restart recovery.
- It reads the transaction logs to reconstruct the sequence of committed and incomplete transactions.
- For committed transactions, redo operations are applied to reapply their changes to the database, ensuring that all committed updates are reflected.
- For incomplete transactions (those in progress at the time of the crash), undo operations are applied to revert any partially applied changes and restore the database to a consistent state.
- Restart recovery ensures that the database maintains its integrity and consistency despite the interruption caused by the system failure.

## Recovery after a System Crash:

Checkpoints are performed as before, except that the checkpoint log record is now of the form

**< checkpoint L>** where L is the list of transactions active at the time of the checkpoint

We assume no updates are in progress while the checkpoint is carried out

Recovery actions, when the database system is restarted after a crash, take place in two phases:

1) REDO PHASE 2) UNDO PHASE

**REDO Phase:**

In the redo phase, the system replays updates of all transactions by scanning the log forward from the last checkpoint. The log records that are replayed include log records for transactions that were rolled back before system crash, and those that had not committed when the system crash occurred.

This phase also determines all transactions that were incomplete at the time of the crash, and must therefore be rolled back.

**The specific steps taken while scanning the log are as follows:**

a. The list of transactions to be rolled back, undo-list, is initially set to the list L in the <checkpoint L> log record.

b. Whenever a normal log record of the form <Ti , Xj , V1, V2>, or a redo-only log record of the form <Ti , Xj , V2> is encountered, the operation is redone; that is, the value V2 is written to data item Xj .

c. Whenever a log record of the form <Ti start> is found, Ti is added to undo-list.

d. Whenever a log record of the form <Ti abort> or <Ti commit> is found, Ti is removed from undo-list.

At the end of the redo phase, undo-list contains the list of all transactions that are incomplete, that is, they neither committed nor completed rollback before the crash.

**UNDO Phase:**

In the undo phase, the system rolls back all transactions in the undo-list. It performs rollback by scanning the log backward from the end.

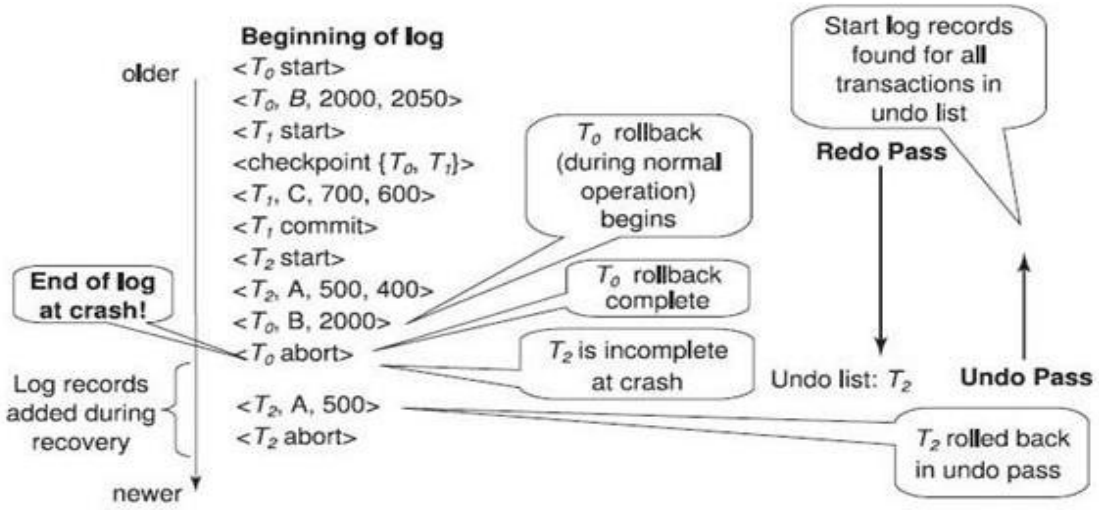
a. Whenever it finds a log record belonging to a transaction in the undo list, it performs undo actions just as if the log record had been found during the rollback of a failed transaction.

b. When the system finds a <Ti start> log record for a transaction Ti in undo-list, it writes a <Ti abort> log record to the log, and removes Ti from undo-list.

c. The undo phase terminates once undo-list becomes empty, that is, the system has found <Tistart>log records for all transactions that were initially in undo-list.

After the undo phase of recovery terminates, normal transaction processing can resume.

**EXAMPLE**




Example of logged actions, and actions during recovery.

The above figure shows an example of actions logged during normal operation, and actions performed during failure recovery. In the log shown in the figure, transaction T1 had committed, and transaction T0 had been completely rolled back, before the system crashed. Observe how the value of data item B is restored during the rollback of T0. Observe also the checkpoint record, with the list of active transactions containing T0 and T1.

When recovering from a crash, in the redo phase, the system performs a redo of all operations after the last checkpoint record. In this phase, the list undo-list initially contains T0 and T1; T1 is removed first when its commit log record is found, while T2 is added when its start log record is found. Transaction T0 is removed from undo-list when its abort log record is found, leaving only T2 in undo-list. The undo phase scans the log backwards from the end, and when it finds a log record of T2 updating A, the old value of A is restored, and a redo-only log record written to the log. When the start record for T2 is found, an abort record is added for T2. Since undo-list contains no more transactions, the undo phase terminates, completing recovery.