Unit 5 – Transaction Management, Concurrency control and Recovery system

Transaction concept and ACID properties:

- When the data of users is stored in a database, that data needs to be accessed and modified from time to time.
- This task should be performed with a specified set of rules and in a systematic way to maintain the consistency and integrity of the data present in a database.
- In DBMS, this task is called a transaction.
- It is similar to a bank transaction, where the user requests to withdraw some amount of money from his account.
- Subsequently, several operations take place such as fetching the user's balance from the database, subtracting the desired amount from it, and updating the user's account balance.
- This series of operations can be called a transaction.
- Transactions are very common in DBMS.

What does a Transaction mean in DBMS?

- Transaction in Database Management Systems (DBMS) can be defined as a set of logically related operations.
- It is the result of a request made by the user to access the contents of the database and perform operations on it.
- It consists of various operations and has various states in its completion journey.
- It also has some specific properties that must be followed to keep the database consistent.

Operations of Transaction

- A user can make different types of requests to access and modify the contents of a database.
- So, we have different types of operations relating to a transaction.
- They are discussed as follows:

i) Read(X)

- A read operation is used to read the value of X from the database and store it in a buffer in the main memory for further actions such as displaying that value.
- Such an operation is performed when a user wishes just to see any content of the database and not make any changes to it.
- For example, when a user wants to check his/her account's balance, a read operation would be performed on user's account balance from the database.

ii) Write(X)

- A write operation is used to write the value to the database from the buffer in the main memory.
- For a write operation to be performed, first a read operation is performed to bring its value in buffer, and then some changes are made to it.
- e.g. some set of arithmetic operations are performed on it according to the user's request, then to store the modified value back in the database, a write operation is performed.
- For example, when a user requests to withdraw some money from his account, his account balance is fetched from the database using a read operation, then the amount to be deducted from the account is subtracted from this value, and then the obtained value is stored back in the database using a write operation.

iii) Commit

- This operation in transactions is used to maintain integrity in the database.
- Due to some failure of power, hardware, or software, etc., a transaction might get interrupted before all its operations are completed.
- This may cause ambiguity in the database, i.e. it might get inconsistent before and after the transaction.
- To ensure that further operations of any other transaction are performed only after work of the current transaction is done, a

commit operation is performed to the changes made by a transaction permanently to the database.

iv) Rollback

- This operation is performed to bring the database to the last saved state when any transaction is interrupted in between due to any power, hardware, or software failure.
- In simple words, it can be said that a rollback operation does undo the operations of transactions that were performed before its interruption to achieve a safe state of the database and avoid any kind of ambiguity or inconsistency.

Properties of Transaction - ACID

- As transactions deal with accessing and modifying the contents of the database, they must have some basic properties which help maintain the consistency and integrity of the database before and after the transaction.
- Transactions follow 4 properties, namely, Atomicity, Consistency, Isolation, and Durability.
- Generally, these are referred to as ACID properties of transactions in DBMS.
- ACID is the acronym used for transaction properties.
- A brief description of each property of the transaction is as follows.

i) Atomicity

- This property ensures that either all operations of a transaction are executed or it is aborted.
- In any case, a transaction can never be completed partially.
- Each transaction is treated as a single unit (like an atom).
- Atomicity is achieved through commit and rollback operations, i.e. changes are made to the database only if all operations related to a transaction are completed, and if it gets interrupted, any changes made are rolled back using rollback operation to bring the database to its last saved state.

ii) Consistency

- This property of a transaction keeps the database consistent before and after a transaction is completed.
- Execution of any transaction must ensure that after its execution, the database is either in its prior stable state or a new stable state.
- In other words, the result of a transaction should be the transformation of a database from one consistent state to another consistent state.
- Consistency, here means, that the changes made in the database are a result of logical operations only which the user desired to perform and there is not any ambiguity.

iii) Isolation

- This property states that two transactions must not interfere with each other, i.e. if some data is used by a transaction for its execution, then any other transaction can not concurrently access that data until the first transaction has completed.
- It ensures that the integrity of the database is maintained and we don’t get any ambiguous values.
- Thus, any two transactions are isolated from each other.
- This property is enforced by the concurrency control subsystem of DBMS.

iv) Durability

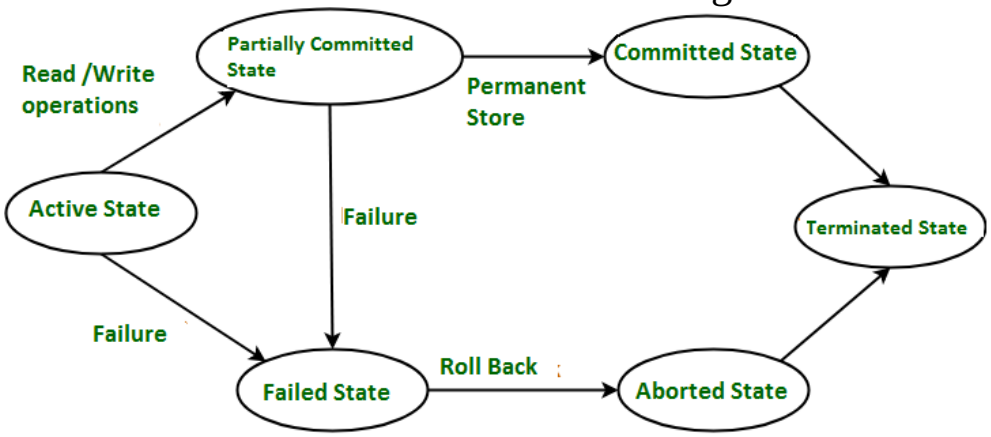
- This property ensures that the changes made to the database after a transaction is completely executed, are durable.
- It indicates that permanent changes are made by the successful execution of a transaction.
- In the event of any system failures or crashes, the consistent state achieved after the completion of a transaction remains intact.
- The recovery subsystem of DBMS is responsible for enforcing this property.

Transaction States in DBMS

States through which a transaction goes during its lifetime. These are the states which tell about the current state of the Transaction and also tell how we will further do the processing in the transactions. These

states govern the rules which decide the fate of the transaction whether it will commit or abort.

They also use **Transaction log.** Transaction log is a file maintain by recovery management component to record all the activities of the transaction. After commit is done transaction log file is removed.



Transaction States in DBMS

These are different types of Transaction States :

Active State
When the instructions of the transaction are running then the transaction is in active state. If all the 'read and write' operations are performed without any error then it goes to the "partially committed state"; if any instruction fails, it goes to the "failed state".

Partially Committed
After completion of all the read and write operation the changes are made in main memory or local buffer. If the changes are made permanent on the DataBase then the state will change to "committed state" and in case of failure it will go to the "failed state".

Failed State
When any instruction of the transaction fails, it goes to the "failed

state" or if failure occurs in making a permanent change of data on Data Base.

Aborted State
After having any type of failure the transaction goes from "failed state" to "aborted state" and since in previous states, the changes are only made to local buffer or main memory and hence these changes are deleted or rolled-back.

Committed State
It is the state when the changes are made permanent on the Data Base and the transaction is complete and therefore terminated in the "terminated state".

Terminated State
If there isn't any roll-back or the transaction comes from the "committed state", then the system is consistent and ready for new transaction and the old transaction is terminated.

Implementation of atomicity and durability:

- Transactions can be incomplete for three kinds of reasons.
- First, a transaction can be aborted, or terminated unsuccessfully, by the DBMS because some anomaly arises during execution. If a transaction is aborted by the DBMS for SOlne internal reason, it is automatically restarted and executed anew.
- Second, the system may crash (e.g., because the power supply is interrupted) while one or more transactions are in progress.
- Third, a transaction may encounter an unexpected situation (for example, read an unexpected data value or be unable to access some disk) and decide to abort (i.e., terminate itself).
- Of course, since users think of transactions; being atomic, a transaction that is interrupted in the middle may leave the database in an inconsistent state.

- Therefore, a DBMS must find a way to remove the effects of partial transactions from the database.
- That is, it must ensure transaction atomicity: Either all of a transaction's actions are carried out or none are.
- A DBMS ensures transaction atomicity by undoing the actions of incomplete transactions.
- This means that users can ignore incomplete transactions in thinking about how the database is modified by transactions over time.
- To be able to do this, the DBMS maintains a record, called the log. of all writes to the database.
- The log is also used to ensure durability: If the system crashes before the changes made by a completed transaction are written to disk, the log is used to remember and restore these changes when the system restarts.
- The DBMS component that ensures atomicity and durability is called the recovery manager.

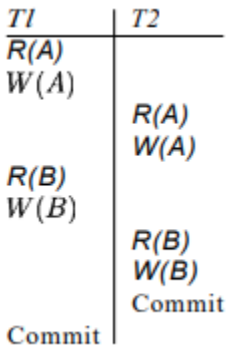
CONCURRENT EXECUTION OF TRANSACTIONS:

- Now that we have introduced the concept of a schedule, we have a convenient way to describe interleaved executions of transactions.
- The DBMS interleaves the actions of different transactions to improve performance, but not all interleavings should be allowed.
- Ensuring transaction isolation while permitting such concurrent execution is difficult but necessary for performance reasons.
- First, while one transaction is waiting for a page to be read in from disk, the CPU can process another transaction.
- This is because I/O activity can be done in parallel with CPU activity in a computer.
- Overlapping I/O and CPU activity reduces the amount of time disks and processors are idle and increases system throughput (the average number of transactions completed in a given time).

- Second, interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly.
- In serial execution, a short transaction could get stuck behind a long transaction, leading to unpredictable delays in response time, or average time taken to complete a transaction.

Serializability:

- 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 the below 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 T1; T2.



- Executing transactions serially in different orders may produce different results, but all are presumed to be acceptable: the DBMS makes no guarantees about which of them will be the outcome of an interleaved execution.
- To see this, note that the example transactions can be interleaved.
- This schedule, also serializable, is equivalent to the serial schedule T2; T1.
- If T1 and T2 are submitted concurrently to a DBMS, either of these schedules (among others) could be chosen.
- The preceding definition of a serializable schedule does not cover the case of schedules containing aborted transactions.

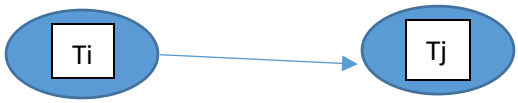
Testing of serializability:

Serialization Graph is used to test the Serializability of a schedule.

Assume a schedule S. For S, we 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:

1. Create a node Ti → Tj if Ti executes write (Q) before Tj executes read (Q).
2. Create a node Ti → Tj if Ti executes read (Q) before Tj executes write (Q).
3. 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 1:

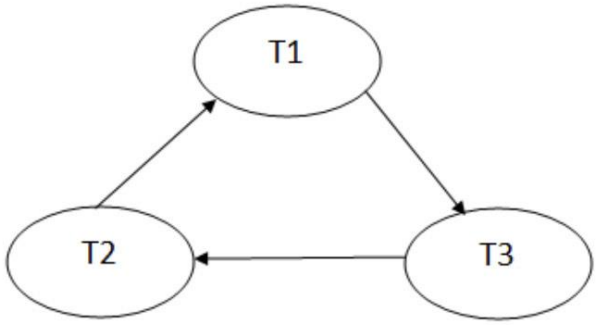
Time

T1	T2	T3
Read(A)		
	Read(B)	
$A := f_1(A)$		
		Read(C)
	$B := f_2(B)$	
	Write(B)	
		$C := f_3(C)$
		Write(C)
Write(A)		
		Read(B)
	Read(A)	
	$A := f_4(A)$	
Read(C)		
	Write(A)	
$C := f_5(C)$		
Write(C)		
		$B := f_6(B)$
		Write(B)

Schedule S1

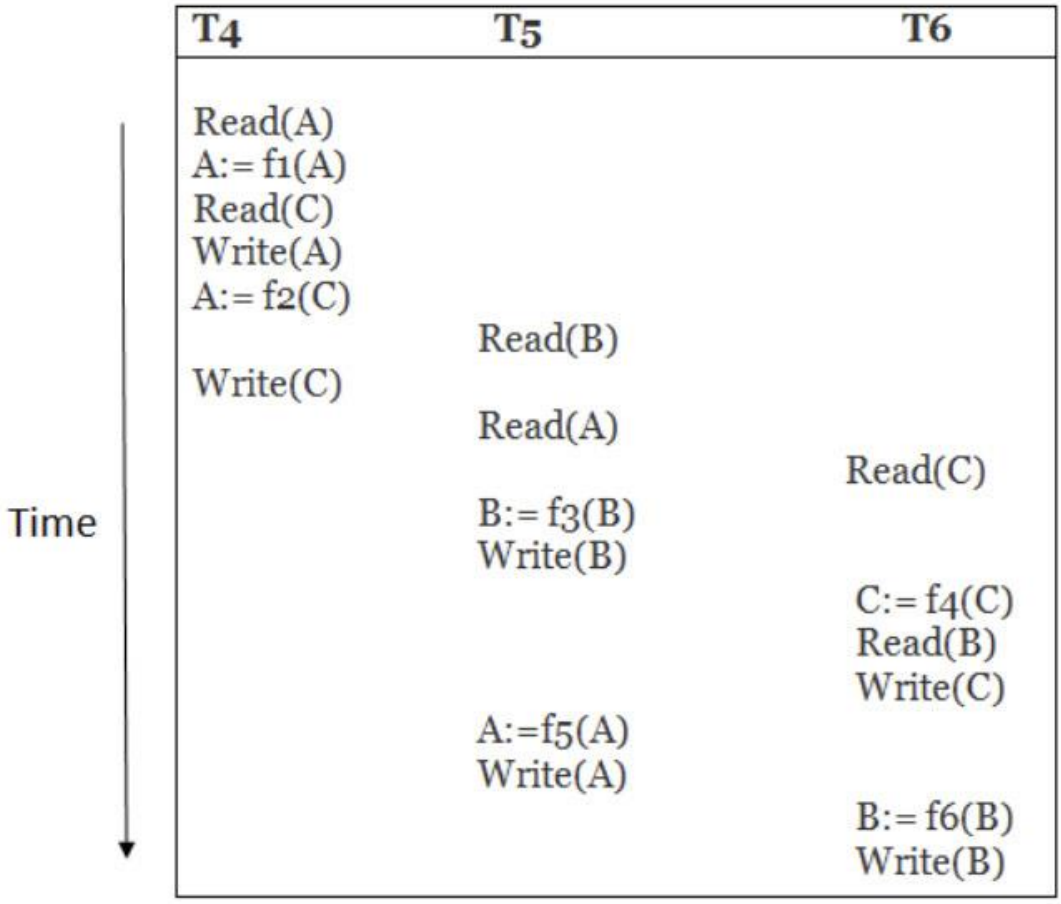
Read(A): In T1, no subsequent writes to A, so no new edges
Read(B): In T2, no subsequent writes to B, so no new edges
Read(C): In T3, no subsequent writes to C, so no new edges
Write(B): B is subsequently read by T3, so add edge T2 → T3
Write(C): C is subsequently read by T1, so add edge T3 → T1
Write(A): 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:



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

Example 2:

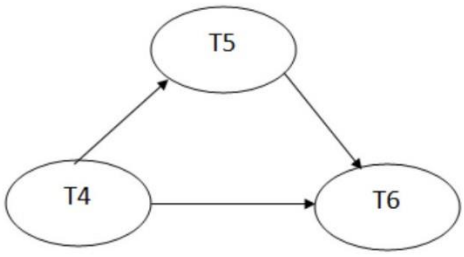


Schedule S2

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:



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

Recoverability:

Recoverability is a property of database systems that ensures that, in the event of a failure or error, the system can recover the database to a consistent state. Recoverability guarantees that all committed transactions are durable and that their effects are permanently stored in the database, while the effects of uncommitted transactions are undone to maintain data consistency.

The recoverability property is enforced through the use of transaction logs, which record all changes made to the database during transaction processing. When a failure occurs, the system uses the log to recover the database to a consistent state, which involves either undoing the effects of uncommitted transactions or redoing the effects of committed transactions.

There are several levels of recoverability that can be supported by a database system:

No-undo logging: This level of recoverability only guarantees that committed transactions are durable, but does not provide the ability to undo the effects of uncommitted transactions.

Undo logging: This level of recoverability provides the ability to undo the effects of uncommitted transactions but may result in the loss of updates made by committed transactions that occur after the failed transaction.

Redo logging: This level of recoverability provides the ability to redo the effects of committed transactions, ensuring that all committed updates are durable and can be recovered in the event of failure.

Undo-redo logging: This level of recoverability provides both undo and redo capabilities, ensuring that the system can recover to a consistent state regardless of whether a transaction has been committed or not.

In addition to these levels of recoverability, database systems may also use techniques such as checkpointing and shadow paging to improve recovery performance and reduce the overhead associated with logging.

Overall, recoverability is a crucial property of database systems, as it ensures that data is consistent and durable even in the event of failures or errors. It is important for database administrators to understand the level of recoverability provided by their system and to configure it appropriately to meet their application's requirements.

As discussed, a transaction may not execute completely due to hardware failure, system crash or software issues. In that case, we have to roll back the failed transaction. But some other transactions may also have used values produced by the failed transaction. So we have to roll back those transactions as well.

Recoverable Schedules:

- Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction T_j is reading value updated or

written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i.

Irrecoverable Schedule: The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. T2 commits. But later on, T1 fails. So we have to rollback T1. Since T2 has read the value written by T1, it should also be rollbacked. But we have already committed that. So this schedule is irrecoverable schedule. When Tj is reading the value updated by Ti and Tj is committed before committing of Ti, the schedule will be irrecoverable.

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
		Commit;		
Failure Point				
Commit;				

Recoverable with Cascading Rollback: The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. But later on, T1 fails. So we have to rollback T1. Since T2 has read the value written by T1, it should also be rollbacked. As it has not committed, we can rollback T2 as well. So it is recoverable with cascading rollback. Therefore, if Tj is reading value updated by Ti and commit of Tj is delayed till commit of Ti, the schedule is called recoverable with cascading rollback.

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
Failure Point				
Commit;				
		Commit;		

Cascadeless Recoverable Rollback: The table below shows a schedule with two transactions, T1 reads and writes A and commits and that value is read by T2. But if T1 fails before commit, no other transaction has read its value, so there is no need to rollback other transaction. So this is a Cascadeless recoverable schedule. So, if Tj reads value updated by Ti only after Ti is committed, the schedule will be cascadeless recoverable.

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
Commit;				
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
		Commit;		

Implementation of isolation:

Isolation is a database-level characteristic that governs how and when modifications are made, as well as whether they are visible to other users, systems, and other databases. One of the purposes of isolation is to allow many transactions to run concurrently without interfering with their execution.

Isolation is a need for database transactional properties. It is the third ACID (Atomicity, Consistency, Isolation, and Durability) standard property that ensures data consistency and accuracy.

Phenomena Defining Isolation Level:

- A transaction that reads data that hasn't yet been committed is said to have performed a **"Dirty Read"**. Imagine that when Transaction 2 receives the modified row, Transaction 1 modifies the row and leaves it uncommitted. Transaction 2 will have read data that was never intended to exist if transaction 1 reverses the change.
- **Non Repeatable Read** occurs when a transaction reads the same row twice and receives a different value each time. Assume that transaction T1 reads data. Because of concurrency, another transaction, T2, modifies and commits the same data. Transaction T1 will get a different value if it reads the same data a second time.
- When two identical queries are run, but the rows returned by the two are different, this phenomenon is known as a **"Phantom Read."** Assume transaction T1 receives a collection of records that meet some search criteria. Transaction T2 now creates some new data that fit the transaction T1 search criteria. Transaction T1 will acquire a different set of rows if it re-executes the statement that reads the rows.

The SQL standard defines four isolation levels based on these phenomena:

Levels of Isolation:

Isolation is divided into four stages. The ability of users to access the same data concurrently is constrained by higher isolation.The greater the isolation degree, the more system resources are required, and the greater the likelihood that database transactions would block one another.

- "**Serializable,**" the highest level, denotes that one transaction must be completed before another can start.
- **Repeatable Reads** allow transactions to be accessed after they have begun, even if they have not completed. This level enables phantom reads or the awareness of inserted or deleted rows even when changes to existing rows are not readable.
- **Read Committed** allows you access to information only after it has been committed to the database.
- **Read Uncommitted** is the lowest level of isolation, allowing access to data before modifications are performed.

Users are more prone to experience read phenomena like uncommitted dependencies, often known as dirty reads, where data is read from a row that has been modified by another user but has not yet been committed to the database, and the lower the isolation level.

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.

Concurrent access is quite easy if all users are just reading data. There is no way they can interfere with one another. Though for any practical Database, it would have a mix of READ and WRITE operations and hence the concurrency is a challenge.

DBMS Concurrency Control is used to address such conflicts, which mostly occur with a multi-user system. Therefore, Concurrency Control is the most important element for proper functioning of a Database Management System where two or more database transactions are executed simultaneously, which require access to the same data.

Potential problems of Concurrency

Here, are some issues which you will likely to face while using the DBMS Concurrency Control method:

- **Lost Updates** occur when multiple transactions select the same row and update the row based on the value selected
- Uncommitted dependency issues occur when the second transaction selects a row which is updated by another transaction (**dirty read**)
- **Non-Repeatable Read** occurs when a second transaction is trying to access the same row several times and reads different data each time.
- **Incorrect Summary issue** occurs when one transaction takes summary over the value of all the instances of a repeated data-item, and second transaction update few instances of that specific data-item. In that situation, the resulting summary does not reflect a correct result.

Why use Concurrency method?

Reasons for using Concurrency control method is DBMS:

- To apply Isolation through mutual exclusion between conflicting transactions
- To resolve read-write and write-write conflict issues
- To preserve database consistency through constantly preserving execution obstructions
- The system needs to control the interaction among the concurrent transactions. This control is achieved using concurrent-control schemes.
- Concurrency control helps to ensure serializability

Example

Assume that two people who go to electronic kiosks at the same time to buy a movie ticket for the same movie and the same show time.

However, there is only one seat left in for the movie show in that particular theatre. Without concurrency control in DBMS, it is possible that both moviegoers will end up purchasing a ticket. However, concurrency control method does not allow this to happen. Both moviegoers can still access information written in the movie seating database. But concurrency control only provides a ticket to the buyer who has completed the transaction process first.

Concurrency Control Protocols

Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose. Following are the Concurrency Control techniques in DBMS:

- Lock-Based Protocols
- Two Phase Locking Protocol
- Timestamp-Based Protocols
- Validation-Based Protocols

Lock-based Protocols

Lock Based Protocols in DBMS is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock. Lock based protocols help to eliminate the concurrency problem in DBMS for simultaneous transactions by locking or isolating a particular transaction to a single user.

A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks in DBMS help synchronize access to the database items by concurrent transactions.

All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

Binary Locks: A Binary lock on a data item can either locked or unlocked states.

Shared/exclusive: This type of locking mechanism separates the locks in DBMS based on their uses. If a lock is acquired on a data item to perform a write operation, it is called an exclusive lock.

1. Shared Lock (S):

A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions. This is because you will never have permission to update data on the data item.

For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account's balance, shared lock prevent it until the reading process is over.

2. Exclusive Lock (X):

With the Exclusive Lock, a data item can be read as well as written. This is exclusive and can't be held concurrently on the same data item. X-lock is requested using lock-x instruction. Transactions may unlock the data item after finishing the 'write' operation.

For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

3. Simplistic Lock Protocol

This type of lock-based protocols allows transactions to obtain a lock on every object before beginning operation. Transactions may unlock the data item after finishing the 'write' operation.

4. Pre-claiming Locking

Pre-claiming lock protocol helps to evaluate operations and create a list of required data items which are needed to initiate an execution process. In the situation when all locks are granted, the transaction executes. After that, all locks release when all of its operations are over.

Starvation

Starvation is the situation when a transaction needs to wait for an indefinite period to acquire a lock.

Following are the reasons for Starvation:

- When waiting scheme for locked items is not properly managed
- In the case of resource leak
- The same transaction is selected as a victim repeatedly

Deadlock

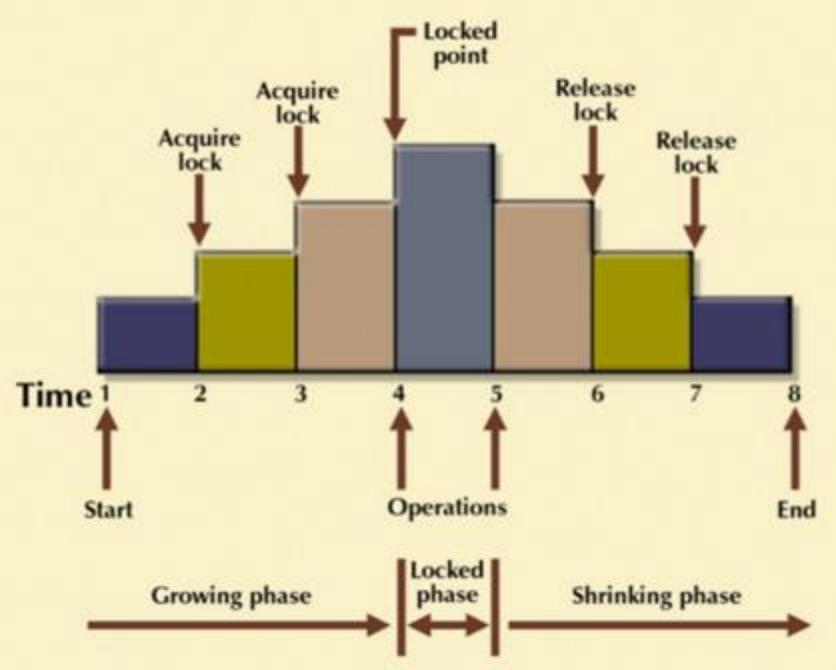
Deadlock refers to a specific situation where two or more processes are waiting for each other to release a resource or more than two processes are waiting for the resource in a circular chain.

Two Phase Locking Protocol

Two Phase Locking Protocol also known as 2PL protocol is a method of concurrency control in DBMS that ensures serializability by applying a lock to the transaction data which blocks other transactions to access the same data simultaneously. Two Phase Locking protocol helps to eliminate the concurrency problem in DBMS.

This locking protocol divides the execution phase of a transaction into three different parts.

- In the first phase, when the transaction begins to execute, it requires permission for the locks it needs.
- The second part is where the transaction obtains all the locks. When a transaction releases its first lock, the third phase starts.
- In this third phase, the transaction cannot demand any new locks. Instead, it only releases the acquired locks.



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

- **Growing Phase**: In this phase transaction may obtain locks but may not release any locks.
- **Shrinking Phase**: In this phase, a transaction may release locks but not obtain any new lock

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

In the above-given diagram, you can see that local and global deadlock detectors are searching for deadlocks and solve them with resuming transactions to their initial states.

Strict Two-Phase Locking Method

Strict-Two phase locking system is almost similar to 2PL. The only difference is that Strict-2PL never releases a lock after using it. It holds all the locks until the commit point and releases all the locks at one go when the process is over.

Centralized 2PL

In Centralized 2 PL, a single site is responsible for lock management process. It has only one lock manager for the entire DBMS.

Primary copy 2PL

Primary copy 2PL mechanism, many lock managers are distributed to different sites. After that, a particular lock manager is responsible for managing the lock for a set of data items. When the primary copy has been updated, the change is propagated to the slaves.

Distributed 2PL

In this kind of two-phase locking mechanism, Lock managers are distributed to all sites. They are responsible for managing locks for data at that site. If no data is replicated, it is equivalent to primary copy 2PL. Communication costs of Distributed 2PL are quite higher than primary copy 2PL

Timestamp-based Protocols

Timestamp based Protocol in DBMS is an algorithm which uses the System Time or Logical Counter as a timestamp to serialize the execution of concurrent transactions. The Timestamp-based protocol ensures that every conflicting read and write operations are executed in a timestamp order.

The older transaction is always given priority in this method. It uses system time to determine the time stamp of the transaction. This is the most commonly used concurrency protocol.

Lock-based protocols help you to manage the order between the conflicting transactions when they will execute. Timestamp-based protocols manage conflicts as soon as an operation is created.

Example:

Suppose there are there transactions T1, T2, and T3.

T1 has entered the system at time 0010

T2 has entered the system at 0020

T3 has entered the system at 0030

Priority will be given to transaction T1, then transaction T2 and lastly Transaction T3.

Advantages:

- Schedules are serializable just like 2PL protocols
- No waiting for the transaction, which eliminates the possibility of deadlocks!

Disadvantages:

Starvation is possible if the same transaction is restarted and continually aborted

Validation Based Protocol

Validation based Protocol in DBMS also known as Optimistic Concurrency Control Technique is a method to avoid concurrency in transactions. In this protocol, the local copies of the transaction data are updated rather than the data itself, which results in less interference while execution of the transaction.

The Validation based Protocol is performed in the following three phases:

1. Read Phase
2. Validation Phase
3. Write Phase

Read Phase

In the Read Phase, the data values from the database can be read by a transaction but the write operation or updates are only applied to the local data copies, not the actual database.

Validation Phase

In Validation Phase, the data is checked to ensure that there is no violation of serializability while applying the transaction updates to the database.

Write Phase

In the Write Phase, the updates are applied to the database if the validation is successful, else; the updates are not applied, and the transaction is rolled back.

Characteristics of Good Concurrency Protocol

An ideal concurrency control DBMS mechanism has the following objectives:

- Must be resilient to site and communication failures.
- It allows the parallel execution of transactions to achieve maximum concurrency.
- Its storage mechanisms and computational methods should be modest to minimize overhead.
- It must enforce some constraints on the structure of atomic actions of transactions.

Advantages of Concurrency

In general, concurrency means, that more than one transaction can work on a system. The advantages of a concurrent system are:

- **Waiting Time:** It means if a process is in a ready state but still the process does not get the system to get execute is called waiting time. So, concurrency leads to less waiting time.
- **Response Time:** The time wasted in getting the response from the cpu for the first time, is called response time. So, concurrency leads to less Response Time.
- **Resource Utilization:** The amount of Resource utilization in a particular system is called Resource Utilization. Multiple transactions can run parallel in a system. So, concurrency leads to more Resource Utilization.
- **Efficiency:** The amount of output produced in comparison to given input is called efficiency. So, Concurrency leads to more Efficiency.

Disadvantages of Concurrency

- **Overhead:** Implementing concurrency control requires additional overhead, such as acquiring and releasing locks on database objects. This overhead can lead to slower performance and increased resource consumption, particularly in systems with high levels of concurrency.
- **Deadlocks:** Deadlocks can occur when two or more transactions are waiting for each other to release resources, causing a circular dependency that can prevent any of the transactions from completing. Deadlocks can be difficult to detect and resolve, and can result in reduced throughput and increased latency.
- **Reduced concurrency:** Concurrency control can limit the number of users or applications that can access the database simultaneously. This can lead to reduced concurrency and slower performance in systems with

high levels of concurrency.

- **Complexity:** Implementing concurrency control can be complex, particularly in distributed systems or in systems with complex transactional logic. This complexity can lead to increased development and maintenance costs.
- **Inconsistency:** In some cases, concurrency control can lead to inconsistencies in the database. For example, a transaction that is rolled back may leave the database in an inconsistent state, or a long-running transaction may cause other transactions to wait for extended periods, leading to data staleness and reduced accuracy.

Crash recovery:

Database Systems like any other computer system, are subject to failures but the data stored in them must be available as and when required. When a database fails it must possess the facilities for fast recovery. It must also have atomicity i.e. either transactions are completed successfully and committed (the effect is recorded permanently in the database) or the transaction should have no effect on the database.

Types of Recovery Techniques in DBMS

Database recovery techniques are used in database management systems (DBMS) to restore a database to a consistent state after a failure or error has occurred. The main goal of recovery techniques is to ensure data integrity and consistency and prevent data loss.

There are mainly two types of recovery techniques used in DBMS

- **Rollback/Undo Recovery Technique**
- **Commit/Redo Recovery Technique**

Rollback/Undo Recovery Technique

The rollback/undo recovery technique is based on the principle of backing out or undoing the effects of a transaction that has not been completed successfully due to a system failure or error. This technique is accomplished by undoing the changes made by the transaction using the log records stored in the transaction log. The transaction log contains a record of all the transactions that have been performed on the database. The system uses the log records to undo the changes made by the failed transaction and restore the database to its previous state.

Commit/Redo Recovery Technique

The commit/redo recovery technique is based on the principle of reapplying the changes made by a transaction that has been completed successfully to the database. This technique is accomplished by using the log records stored in the transaction log to redo the changes made by the transaction that was in progress at the time of the failure or error. The system uses the log records to reapply the changes made by the transaction and restore the database to its most recent consistent state.

In addition to these two techniques, there is also a third technique called checkpoint recovery.

Checkpoint Recovery is a technique used to reduce the recovery time by periodically saving the state of the database in a checkpoint file. In the event of a failure, the system can use the checkpoint file to restore the database to the most recent consistent state before the failure occurred, rather than going through the entire log to recover the database.

Overall, recovery techniques are essential to ensure data consistency and availability in Database Management System, and each technique has its own advantages and limitations that must be considered in the design of a recovery system.

Database Systems

There are both automatic and non-automatic ways for both, backing up data and recovery from any failure situations. The techniques used to recover lost data due to system crashes, transaction errors, viruses, catastrophic failure, incorrect command execution, etc. are database recovery techniques. So to prevent data loss recovery techniques based on deferred updates and immediate updates or backing up data can be used. Recovery techniques are heavily dependent upon the existence of a special file known as a **system log**. It contains information about the start and end of each transaction and any updates which occur during the **transaction**. The log keeps track of all transaction operations that affect the values of database items. This information is needed to recover from transaction failure.

- **The log is kept on disk start_transaction(T):** This log entry records that transaction T starts the execution.
- **read_item(T, X):** This log entry records that transaction T reads the value of database item X.
- **write_item(T, X, old_value, new_value):** This log entry records that transaction T changes the value of the database item X from old_value to new_value. The old value is sometimes known as a before an image of X, and the new value is known as an afterimage of X.
- **commit(T):** This log entry records that transaction T has completed all accesses to the database successfully and its effect can be committed (recorded permanently) to the database.
- **abort(T):** This records that transaction T has been aborted.
- **checkpoint:** A checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in a consistent state, and all the transactions were committed.

A transaction T reaches its **commit** point when all its operations that access the database have been executed successfully i.e. the transaction has reached the point at which it will not **abort** (terminate without completing). Once committed, the transaction is permanently recorded

in the database. Commitment always involves writing a commit entry to the log and writing the log to disk. At the time of a system crash, the item is searched back in the log for all transactions T that have written a start_transaction(T) entry into the log but have not written a commit(T) entry yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process.

- **Undoing:** If a transaction crashes, then the recovery manager may undo transactions i.e. reverse the operations of a transaction. This involves examining a transaction for the log entry write_item(T, x, old_value, new_value) and setting the value of item x in the database to old-value. There are two major techniques for recovery from non-catastrophic transaction failures: deferred updates and immediate updates.
- **Deferred Update:** This technique does not physically update the database on disk until a transaction has reached its commit point. Before reaching commit, all transaction updates are recorded in the local transaction workspace. If a transaction fails before reaching its commit point, it will not have changed the database in any way so UNDO is not needed. It may be necessary to REDO the effect of the operations that are recorded in the local transaction workspace, because their effect may not yet have been written in the database. Hence, a deferred update is also known as the **No-undo/redo algorithm.**
  - **Immediate Update:** In the immediate update, the database may be updated by some operations of a transaction before the transaction reaches its commit point. However, these operations are recorded in a log on disk before they are applied to the database, making recovery still possible. If a transaction fails to reach its commit point, the effect of its operation must be undone i.e. the transaction must be rolled back hence we require both undo and redo. This technique is known as **undo/redo algorithm.**
  - **Caching/Buffering:** In this one or more disk pages that include data items to be updated are cached into main memory buffers and

then updated in memory before being written back to disk. A collection of in-memory buffers called the DBMS cache is kept under the control of DBMS for holding these buffers. A directory is used to keep track of which database items are in the buffer. A dirty bit is associated with each buffer, which is 0 if the buffer is not modified else 1 if modified.

- **Shadow Paging:** It provides atomicity and durability. A directory with n entries is constructed, where the ith entry points to the ith database page on the link. When a transaction began executing the current directory is copied into a shadow directory. When a page is to be modified, a shadow page is allocated in which changes are made and when it is ready to become durable, all pages that refer to the original are updated to refer new replacement page.
- **Backward Recovery:** The term "**Rollback**" and "**UNDO**" can also refer to backward recovery. When a backup of the data is not available and previous modifications need to be undone, this technique can be helpful. With the backward recovery method, unused modifications are removed and the database is returned to its prior condition. All adjustments made during the previous traction are reversed during the backward recovery. In other words, it reprocesses valid transactions and undoes the erroneous database updates.
- **Forward Recovery:** "Roll forward "and "REDO" refers to forwarding recovery. When a database needs to be updated with all changes verified, this forward recovery technique is helpful. Some failed transactions in this database are applied to the database to roll those modifications forward. In other words, the database is restored using preserved data and valid transactions counted by their past saves.

Backup Techniques

There are different types of Backup Techniques. Some of them are listed below.

- **Full database Backup:** In this full database including data and database, Meta information needed to restore the whole database, including full-text catalogs are backed up in a predefined time series.
- **Differential Backup:** It stores only the data changes that have occurred since the last full database backup. When some data has changed many times since the last full database backup, a differential backup stores the most recent version of the changed data. For this first, we need to restore a full database backup.
- **Transaction Log Backup:** In this, all events that have occurred in the database, like a record of every single statement executed is backed up. It is the backup of transaction log entries and contains all transactions that had happened to the database. Through this, the database can be recovered to a specific point in time. It is even possible to perform a backup from a transaction log if the data files are destroyed and not even a single committed transaction is lost.

Remote Backup Systems

Remote backup systems provide a wide range of availability, allowing the transaction processing to continue even if the primary site is destroyed by a fire, flood or earthquake.Data and log records from a primary site are continuously backed up into a remote backup site.

One can achieve ‘wide range availability’ of data 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 duplicated.
The remote site is also called ‘**secondary site**’.

The remote site must be **synchronized** with the primary site, as updates are performed at the primary.
In designing a remote backup system, the following points are important.
a) Detection of failure: It is important for the remote backup system to detect when the primary has failed.

b) Transfer of control: When the primary site fails, the backup site takes over the processing and becomes the new primary site.

c) Time to recover: If the log at the remote backup becomes large, recovery will take a long time.

d) Time to commit: To ensure that the updates of a committed transaction are durable, a transaction should not be announced committed until its log records have reached the backup site.