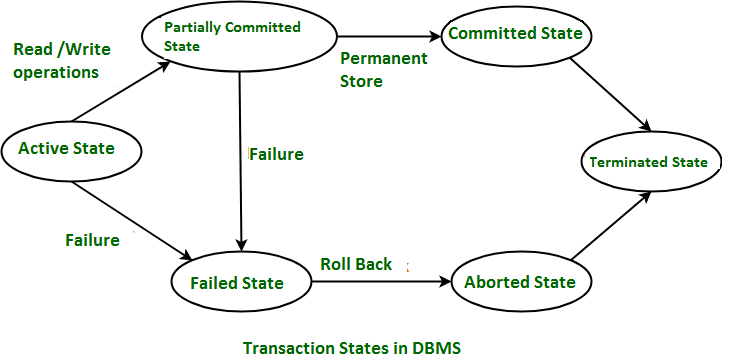
**1. Define Transaction State. Draw State diagram of a transaction and explain.**

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.

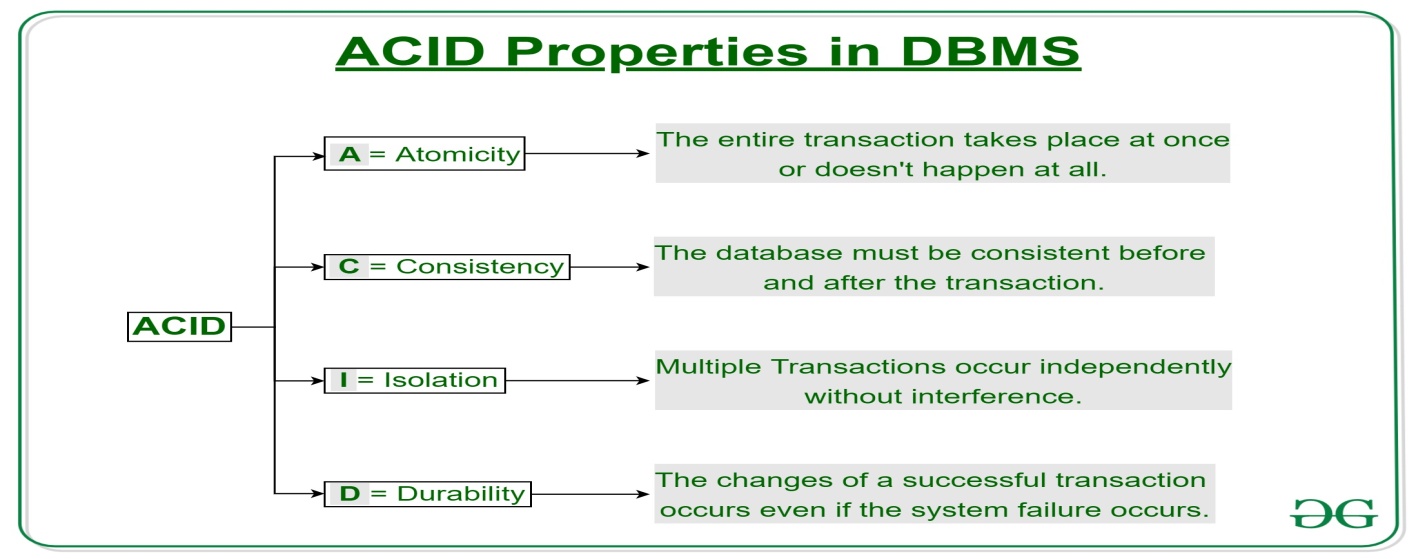


These are different types of Transaction States :

1. **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”.
2. **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”.
3. **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.
4. **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.
5. **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”.
6. **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.

**2. Explain about ACID properties.**

A [**transaction**](https://www.geeksforgeeks.org/sql-transactions/) is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.   
In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.



### ****Atomicity:****

By this, we mean that either the entire transaction takes place at once or doesn’t happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.   
—**Abort**: If a transaction aborts, changes made to the database are not visible.   
—**Commit**: If a transaction commits, changes made are visible.   
Atomicity is also known as the ‘All or nothing rule’.

Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.



If the transaction fails after completion of **T1** but before completion of **T2**.( say, after **write(X)** but before **write(Y)**), then the amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in its entirety in order to ensure the correctness of the database state.

### Consistency:

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above,   
The total amount before and after the transaction must be maintained.   
Total **before T** occurs = **500 + 200 = 700**.   
Total **after T occurs** = **400 + 300 = 700**.   
Therefore, the database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result, T is incomplete.

### Isolation:

This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.   
Let **X**=500,  **Y** =500.   
Consider two transactions **T** and **T”.**



Suppose **T** has been executed till **Read (Y)** and then **T’’** starts. As a result, interleaving of operations takes place due to which **T’’** reads the correct value of **X** but the incorrect value of **Y** and sum computed by   
**T’’: (X+Y=50, 000+500=50, 500)**

is thus not consistent with the sum at end of the transaction:   
**T: (X+Y = 50, 000 + 450 = 50, 450)**.   
This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

### Durability:

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

**Some important points:**

| **Property** | **Responsibility for maintaining properties** |
| --- | --- |
| Atomicity | Transaction Manager |
| Consistency | Application programmer |
| Isolation | Concurrency Control Manager |
| Durability | Recovery Manager |

The **ACID** properties, in totality, provide a mechanism to ensure the correctness and consistency of a database in a way such that each transaction is a group of operations that acts as a single unit, produces consistent results, acts in isolation from other operations, and updates that it makes are durably stored.

 Advantages of ACID Properties in DBMS:

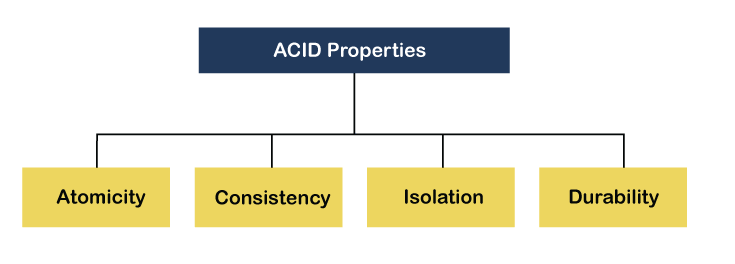
1. Data Consistency: ACID properties ensure that the data remains consistent and accurate after any transaction execution.
2. Data Integrity: ACID properties maintain the integrity of the data by ensuring that any changes to the database are permanent and cannot be lost.
3. Concurrency Control: ACID properties help to manage multiple transactions occurring concurrently by preventing interference between them.
4. Recovery: ACID properties ensure that in case of any failure or crash, the system can recover the data up to the point of failure or crash.

### Disadvantages of ACID Properties in DBMS:

1. Performance: The ACID properties can cause a performance overhead in the system, as they require additional processing to ensure data consistency and integrity.
2. Scalability: The ACID properties may cause scalability issues in large distributed systems where multiple transactions occur concurrently.
3. Complexity: Implementing the ACID properties can increase the complexity of the system and require significant expertise and resources.  
   Overall, the advantages of ACID properties in DBMS outweigh the disadvantages. They provide a reliable and consistent approach to data
4. management, ensuring data integrity, accuracy, and reliability. However, in some cases, the overhead of implementing ACID properties can cause performance and scalability issues. Therefore, it’s important to balance the benefits of ACID properties against the specific needs and requirements of the system.

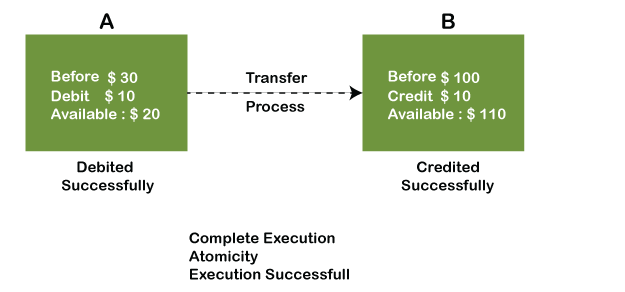
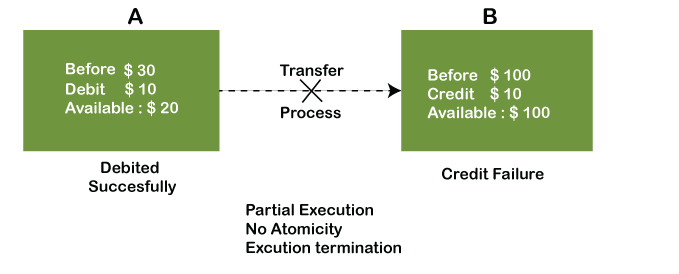
**OR**

The ACID properties are meant for the transaction that goes through a different group of tasks, and there we come to see the role of the ACID properties.



1. **Atomicity**

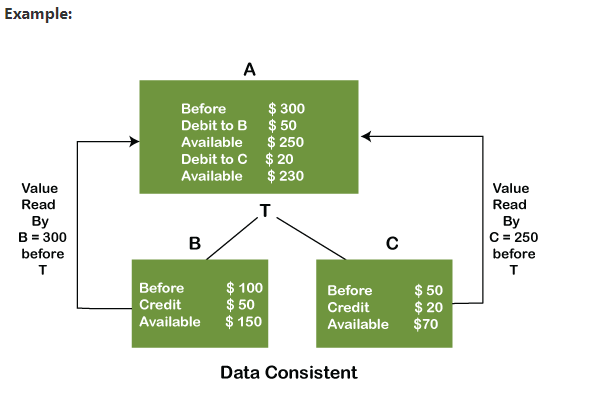
* The term atomicity defines that the data remains atomic.
* It means if any operation is performed on the data, either it should be performed or executed completely or should not be executed at all.
* It means that the operation should not break in between or execute partially.
* In the case of executing operations on the transaction, the operation should be completely executed and not partially.
* **Example:** If Ramu has account A having $30 in his account from which he wishes to send $10 to Sheela's account, which is B.
* In account B, a sum of $ 100 is already present. When $10 will be transferred to account B, the sum will become $110.
* Now, there will be two operations that will take place. One is the amount of $10 that Ramu wants to transfer will be debited from his account A, and the same amount will get credited to account B, i.e., into Sheela's account.
* Now, what happens - the first operation of debit executes successfully, but the credit operation, however, fails. Thus, in Ramu's account A, the value becomes $20, and to that of Sheela's account, it remains $100 as it was previously present.
* In the below diagram, it can be seen that after crediting $10, the amount is still $100 in account B. So, it is not an atomic transaction.
* The below image shows that both debit and credit operations are done successfully. Thus the transaction is atomic



* Thus, when the amount loses atomicity, then in the bank systems, this becomes a huge issue, and so the atomicity is the main focus in the bank systems.

**2) Consistency**

* The word **consistency** means that the value should remain preserved always.
* In DBMS, the integrity of the data should be maintained, which means if a change in the database is made, it should remain preserved always.
* In the case of transactions, the integrity of the data is very essential so that the database remains consistent before and after the transaction.
* The data should always be correct.



* In the above figure, there are three accounts, A, B, and C, where A is making a transaction T one by one to both B & C.
* There are two operations that take place, i.e., Debit and Credit. Account A firstly debits $50 to account B, and the amount in account A is read $300 by B before the transaction.
* After the successful transaction T, the available amount in B becomes $150.
* Now, A debits $20 to account C, and that time, the value read by C is $250 (that is correct as a debit of $50 has been successfully done to B).
* The debit and credit operation from account A to C has been done successfully.
* We can see that the transaction is done successfully, and the value is also read correctly. Thus, the data is consistent.

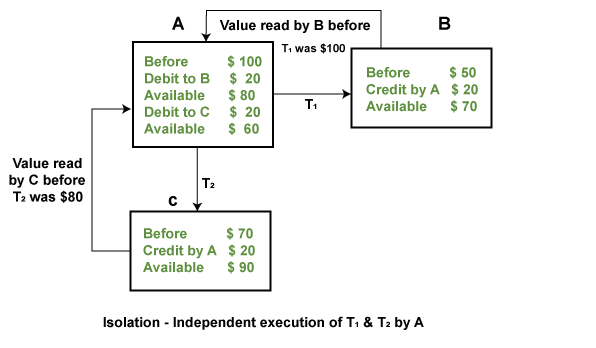
In case the value read by B and C is $300, which means that data is inconsistent because when the debit operation executes, it will not be consistent.

**3) Isolation**

* The term 'isolation' means separation. In DBMS, Isolation is the property of a database where no data should affect the other one and may occur concurrently.
* In short, the operation on one database should begin when the operation on the first database gets complete.
* It means if two operations are being performed on two different databases, they may not affect the value of one another.
* In the case of transactions, when two or more transactions occur simultaneously, the consistency should remain maintained.
* Any changes that occur in any particular transaction will not be seen by other transactions until the change is not committed in the memory.

**Example:**

* If two operations are concurrently running on two different accounts, then the value of both accounts should not get affected.
* The value should remain persistent.
* As you can see in the below diagram, account A is making T1 and T2 transactions to account B and C, but both are executing independently without affecting each other. It is known as Isolation.



4) Durability

* Durability ensures the permanency of something.
* In DBMS, the term durability ensures that the data after the successful execution of the operation becomes permanent in the database.
* The durability of the data should be so perfect that even if the system fails or leads to a crash, the database still survives. However, if gets lost, it becomes the responsibility of the recovery manager for ensuring the durability of the database.
* For committing the values, the COMMIT command must be used every time we make changes.

Therefore, the ACID property of DBMS plays a vital role in maintaining the consistency and availability of data in the database.

**3. Elaborate the concept of concurrent execution of transactions.**

* In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database.
* It means that the same database is executed simultaneously on a multi-user system by different users.
* While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
* The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database.
* Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

**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.
* 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:**

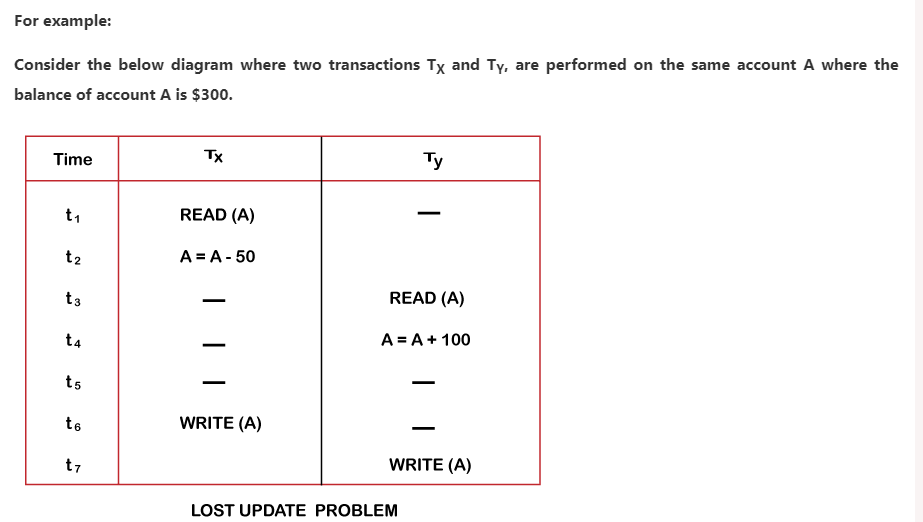
* There may be a mix of transactions running on a system, some short and some long.
* If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction.
* If the transactions are operating on 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.
* The motivation for using concurrent execution in a database is essentially the same as the motivation for using multiprogramming in an operating system.
* The concept of schedules helps to identify concurrent executions that are guaranteed to ensure the isolation property and thus database consistency.

**Problems with Concurrent Execution:**

* In a database transaction, the two main operations are **READ** and **WRITE** operations.
* So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent.
* So, the following problems occur with the Concurrent Execution of the operations:

**Problem 1: Lost Update Problems (W - W Conflict)**

* The problem occurs when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent.

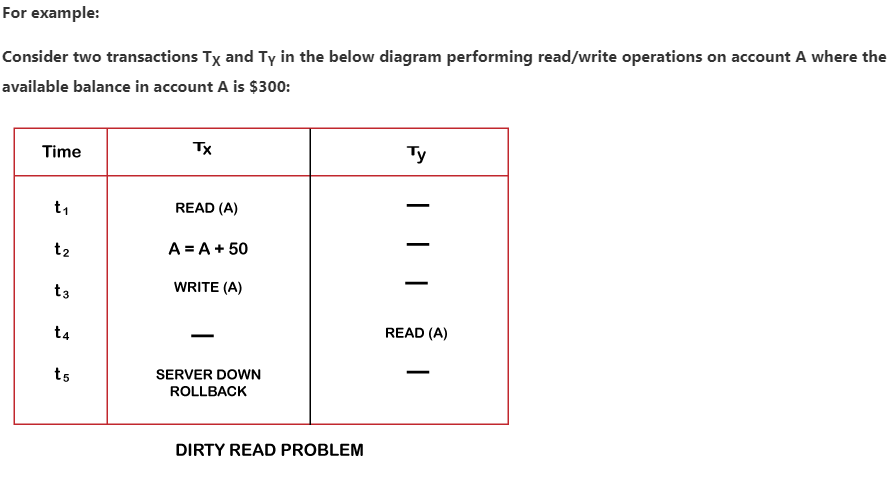


* At time t1, transaction TX reads the value of account A, i.e., $300 (only read).
* At time t2, transaction TX deducts $50 from account A that becomes $250 (only deducted and not updated/write).
* Alternately, at time t3, transaction TY reads the value of account A that will be $300 only because TX didn't update the value yet.
* At time t4, transaction TY adds $100 to account A that becomes $400 (only added but not updated/write).
* At time t6, transaction TX writes the value of account A that will be updated as $250 only, as TY didn't update the value yet.
* Similarly, at time t7, transaction TY writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by TX is lost, i.e., $250 is lost.

Hence data becomes incorrect, and database sets to inconsistent.

**Dirty Read Problems (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.



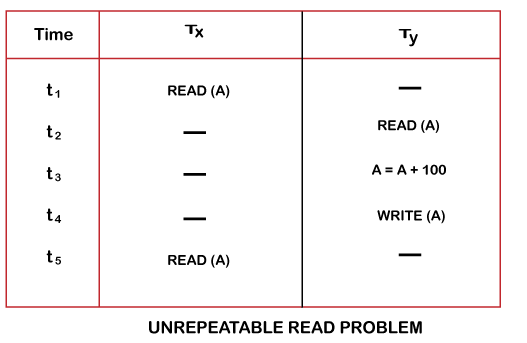
* At time t1, transaction TX reads the value of account A, i.e., $300.
* At time t2, transaction TX adds $50 to account A that becomes $350.
* At time t3, transaction TX writes the updated value in account A, i.e., $350.
* Then at time t4, transaction TY reads account A that will be read as $350.
* Then at time t5, transaction TX rollbacks due to server problem, and the value changes back to $300 (as initially).
* But the value for account A remains $350 for transaction TY as committed, which is the dirty read and therefore known as the Dirty Read Problem.

**Unrepeatable Read Problem (R-R Conflict):**

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

**For example:**

Consider two transactions, TX and TY, performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:



* At time t1, transaction TX reads the value from account A, i.e., $300.
* At time t2, transaction TY reads the value from account A, i.e., $300.
* At time t3, transaction TY updates the value of account A by adding $100 to the available balance, and then it becomes $400.
* At time t4, transaction TY writes the updated value, i.e., $400.
* After that, at time t5, transaction TX reads the available value of account A, and that will be read as $400.
* It means that within the same transaction TX, it reads two different values of account A, i.e., $ 300 initially, and after updation made by transaction TY, it reads $400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.
* Thus, in order to maintain consistency in the database and avoid such problems that take place in concurrent execution, management is needed, and that is where the concept of Concurrency Control comes into role.

**4.Discuss timestamp-based protocol for concurrency control.**

Timestamp-based concurrency control is a method used in database systems to ensure that transactions are executed safely and consistently without conflicts, even when multiple transactions are being processed simultaneously. Timestamp-based concurrency control is a technique used in **Database Management Systems (DBMS)** to manage concurrent transactions while ensuring **serializability**. This approach relies on timestamps to manage and coordinate the execution order of transactions. It assigns a unique timestamp to each transaction and schedules them based on these timestamps to prevent conflicts such as **dirty reads, lost updates, and uncommitted dependencies**.

The Timestamp Ordering Protocol is a method used in database systems to order transactions based on their timestamps. A timestamp is a unique identifier assigned to each transaction, typically determined using the system clock or a logical counter. Transactions are executed in the ascending order of their timestamps, ensuring that older transactions get higher priority.

For example:

* If Transaction T1 enters the system first, it gets a timestamp TS(T1) = 007 (assumption).
* If Transaction T2 enters after T1, it gets a timestamp TS(T2) = 009 (assumption).

This means T1 is “older” than T2 and T1 should execute before T2 to maintain consistency.

### ****Key Concepts****

1. **Timestamp (TS)**
   * Each transaction **T** is assigned a unique timestamp when it begins.
   * This can be based on the system clock or a logical counter.
   * Transactions with **smaller timestamps** are considered **older** and should execute before newer transactions.
2. **Timestamp of Data Items**
   * **Read Timestamp (RTS(X))**: The latest timestamp of a transaction that has successfully read data item **X**.
   * **Write Timestamp (WTS(X))**: The latest timestamp of a transaction that has successfully written data item **X**.

### ****Timestamp Ordering Protocol Rules****

When a transaction **T** attempts to read or write a data item **X**, the following rules apply:

#### **1. Read Operation: T wants to read X (TS(T) < WTS(X))**

* If **TS(T) < WTS(X)**, **T must be aborted** and restarted with a new timestamp to prevent reading an overwritten value.
* Otherwise, the read is allowed, and **RTS(X) is updated**.

#### **2. Write Operation: T wants to write X (TS(T) < RTS(X) or TS(T) < WTS(X))**

* If **TS(T) < RTS(X)**, **T is aborted** to avoid overwriting a newer version that was already read.
* If **TS(T) < WTS(X)**, **T is aborted** because a newer transaction has already written **X**.
* Otherwise, **T writes X**, and **WTS(X) is updated**.

### ****Advantages****

* Ensures **serializability** automatically.
* Avoids **deadlocks** as transactions are never blocked; they are either aborted or executed.
* Suitable for applications where transactions can be easily restarted.

### ****Disadvantages****

* **Frequent transaction restarts** can lead to performance overhead.
* Not suitable for **long-running transactions**.
* If older transactions restart too often, they may experience **starvation**.

### ****Example****

Consider two transactions:

* **T1 (TS = 5)**
* **T2 (TS = 10)**

| **Transaction** | **Action** | **Condition Check** | **Result** |
| --- | --- | --- | --- |
| T1 (TS=5) | Read(X) | TS(T1) > WTS(X) | Allowed |
| T2 (TS=10) | Write(X) | TS(T2) > RTS(X) | Allowed |
| T1 (TS=5) | Write(X) | TS(T1) < WTS(X) | **Aborted** |

Since T1 tries to write after a newer transaction T2 has already written X, T1 is aborted.

**5. Explain the concept of recovery and atomicity.**

* When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items.
* But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.
* Database recovery means recovering the data when it get deleted, hacked or damaged accidentally.
* Atomicity is must whether is transaction is over or not it should reflect in the database permanently or it should not effect the database at all.

**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.

**There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction** −

* Maintaining the logs of each transaction, and writing them onto some stable [storage](https://studyglance.in/dbms/display.php?tno=48&topic=Recovery-and-Atomicity-in-dbms) before actually modifying the database.
* Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated

## 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.

# How Log-Based Recovery Works

## 1. Transaction Logging:

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 <\(T\_i\) , \(X\_j\) , \(V\_1\), \(V\_2\)>, indicating that transaction \(T\_i\) has performed a write on data item \(X\_j\). \(X\_j\) had value \(V\_1\) before the write, and has value \(V\_2\) after the write. Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. Among the types of log records are:

* <\(T\_i\) start>. Transaction Ti has started.
* <\(T\_i\) commit>. Transaction Ti has committed.
* <\(T\_i\) abort>. Transaction Ti has aborted.

## 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

Periodically, the DBMS might decide to take a checkpoint. 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.

# 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.

# Shadow paging - Its Working principle

Shadow Paging is an alternative disk recovery technique to the more common logging mechanisms. It's particularly suitable for database systems. The fundamental concept behind shadow paging is to maintain two page tables during the lifetime of a transaction: the current page table and the shadow page table.

Here's a step-by-step breakdown of the working principle of shadow paging:

## Initialization

When the transaction begins, the database system creates a copy of the current page table. This copy is called the shadow page table.

The actual data pages on disk are not duplicated; only the page table entries are. This means both the current and shadow page tables point to the same data pages initially.

## During Transaction Execution

When a transaction modifies a page for the first time, a copy of the page is made. The current page table is updated to point to this new page.

Importantly, the shadow page table remains unaltered and continues pointing to the original, unmodified page.

Any subsequent changes by the same transaction are made to the copied page, and the current page table continues to point to this copied page.

## On Transaction Commit

Once the transaction reaches a commit point, the shadow page table is discarded, and the current page table becomes the new "truth" for the database state.

The old data pages that were modified during the transaction (and which the shadow page table pointed to) can be reclaimed.

## Recovery after a Crash

If a crash occurs before the transaction commits, recovery is straightforward. Since the original data pages (those referenced by the shadow page table) were never modified, they still represent a consistent database state.

The system simply discards the changes made during the transaction (i.e., discards the current page table) and reverts to the shadow page table.

**6. Explain about transaction processing in detail.**

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.

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.

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

**X's Account**

1. Open\_Account(X)
2. Old\_Balance = X.balance
3. New\_Balance = Old\_Balance - 800
4. X.balance = New\_Balance
5. Close\_Account(X)

**Y's Account**

1. Open\_Account(Y)
2. Old\_Balance = Y.balance
3. New\_Balance = Old\_Balance + 800
4. Y.balance = New\_Balance
5. Close\_Account(Y)

Operations of Transaction:

Following are the main operations of transaction:

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

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

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

1.  R(X);

2.  X = X - 500;

3.  W(X);

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

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

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

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

To solve this problem, we have two important operations:

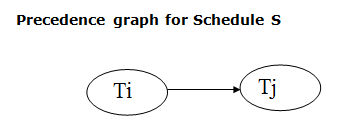
**Commit:** It is used to save the work done permanently.

**Rollback:** It is used to undo the work done.

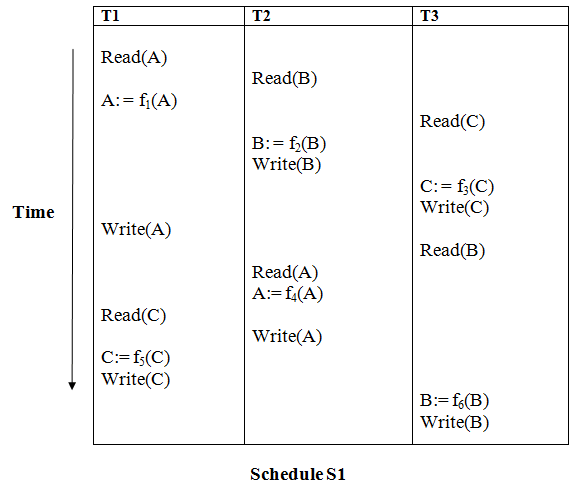
**7. How to test serializability of a schedule? Explain with an example.**

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).  
   

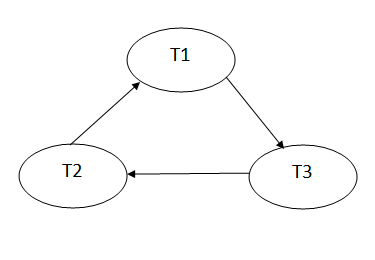
* If a precedence graph contains a single edge Ti → Tj, then all the instructions of Ti are executed before the first instruction of Tj is executed.
* If a precedence graph for schedule S contains a cycle, then S is non-serializable. If the precedence graph has no cycle, then S is known as serializable.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



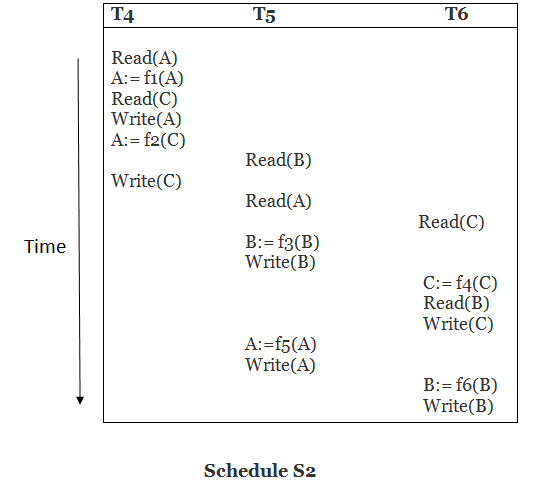
**Explanation:**

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

### Precedence graph for schedule S1:



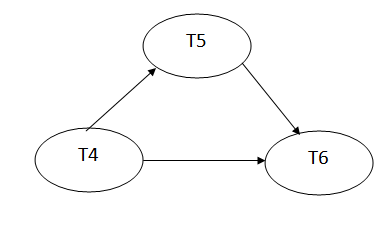
* The precedence graph for schedule S1 contains a cycle that's why Schedule S1 is non-serializable.



**Explanation:**

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

### Precedence graph for schedule S2:



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

**8.Compare deadlock prevention, avoidance, and detection. Write advantages and disadvantages for each approach.**

Deadlocks in **Database Management Systems (DBMS)** occur when multiple transactions compete for resources and end up waiting indefinitely. To manage deadlocks, DBMS employs three primary strategies: **prevention, avoidance, and detection**. Each approach has its own merits and trade-offs.

## **1. Deadlock Prevention**

**Method:**  
Deadlock prevention ensures that at least one of the four necessary conditions for deadlock (Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait) is violated, thereby preventing deadlocks from occurring.

### ****Techniques for Deadlock Prevention in DBMS:****

* **Wait-Die Scheme:**
  + Older transactions can wait for a resource held by a younger transaction.
  + Younger transactions requesting resources held by an older transaction are aborted and restarted.
* **Wound-Wait Scheme:**
  + Older transactions can preempt (force rollback) younger transactions.
  + Younger transactions must wait if they request resources held by older ones.
* **Prevention by Eliminating Hold and Wait:**
  + Transactions must request all required resources at the beginning (Conservative Two-Phase Locking).
  + If all resources are not available, the transaction does not start.
* **Prevention by Eliminating Circular Wait:**
  + Transactions acquire resources in a predefined order, preventing cycles in the wait-for graph.

### ****Advantages of Deadlock Prevention:****

✔ **Ensures deadlocks never occur**, improving system stability.  
✔ **Simple to implement** in some cases, especially with predefined resource ordering.  
✔ **No need for deadlock detection mechanisms**, reducing monitoring overhead.

### ****Disadvantages of Deadlock Prevention:****

❌ **Can significantly reduce concurrency**, as transactions may be forced to wait longer before execution.  
❌ **Increased transaction rollbacks** in schemes like Wait-Die or Wound-Wait.  
❌ **Preemptive approaches can lead to performance degradation** due to frequent transaction aborts.

## **2. Deadlock Avoidance**

**Method:**  
Deadlock avoidance dynamically ensures that the system **never enters an unsafe state** by analyzing resource requests and allocations. This requires advance knowledge of resource needs.

### ****Techniques for Deadlock Avoidance in DBMS:****

* **Wait-for Graph Algorithm:**
  + Maintains a directed graph of waiting transactions.
  + If adding an edge creates a cycle, the transaction request is denied or rolled back.
* **Banker’s Algorithm:**
  + Transactions declare maximum resource needs in advance.
  + The system grants resources only if doing so keeps the state "safe" (i.e., all transactions can eventually complete).

### ****Advantages of Deadlock Avoidance:****

✔ **Higher concurrency than prevention**, since resources are allocated dynamically.  
✔ **Ensures deadlocks do not occur**, unlike detection-based approaches.  
✔ **More flexible than prevention**, as transactions do not have to request all resources at the start.

### ****Disadvantages of Deadlock Avoidance:****

❌ **Requires prior knowledge of resource needs**, which is not always feasible in real-world applications.  
❌ **Increases system overhead**, as the DBMS must continuously monitor and analyze the state of transactions.  
❌ **Not practical for large-scale dynamic systems**, where resource needs change frequently.

## **3. Deadlock Detection**

**Method:**  
Instead of preventing or avoiding deadlocks, detection allows them to occur but **identifies and resolves them** afterward.

### ****Techniques for Deadlock Detection in DBMS:****

* **Wait-for Graph (WFG) Analysis:**
  + Constructs a graph where transactions are nodes and edges represent resource waits.
  + If a cycle is found, a deadlock is detected, and a victim transaction is selected for rollback.
* **Timeout-Based Detection:**
  + If a transaction waits too long for a resource, it is assumed to be in a deadlock and is aborted.

### ****Advantages of Deadlock Detection:****

✔ **Allows maximum concurrency**, as no restrictive policies are applied before execution.  
✔ **No need for transactions to declare resource needs in advance.**  
✔ **Efficient handling of occasional deadlocks**, making it suitable for dynamic environments.

### ****Disadvantages of Deadlock Detection:****

❌ **Requires periodic checks**, which increases computational overhead.  
❌ **Deadlocks still occur**, requiring recovery mechanisms like transaction rollback.  
❌ **Transaction rollbacks may result in loss of progress**, leading to performance degradation.

## **Comparison Table**

| **Approach** | **Method** | **Advantages** | **Disadvantages** |
| --- | --- | --- | --- |
| **Deadlock Prevention** | Prevents deadlocks by eliminating necessary conditions (e.g., preemption, circular wait). | - Guarantees deadlock-free operation.  - Simple implementation in some cases.  - No need for continuous monitoring. | - Reduces concurrency, causing delays.  - May require excessive rollbacks.  - Predefined resource ordering is impractical. |
| **Deadlock Avoidance** | Dynamically avoids unsafe states using algorithms like Wait-for Graph or Banker’s Algorithm. | - Allows higher concurrency than prevention.  - Ensures deadlocks do not occur.  - More flexible than prevention. | - Requires prior knowledge of resource needs.  - High system overhead due to continuous monitoring.  - Not practical for dynamic workloads. |
| **Deadlock Detection** | Allows deadlocks but detects them using Wait-for Graph or timeout-based methods. | - Maximizes concurrency.  - No need for predefined resource ordering.  - Efficient for environments with rare deadlocks. | - Deadlocks still occur, requiring resolution.  - Transaction rollbacks can be costly.  - Requires periodic monitoring, increasing overhead. |

**9. Outline the lock based concurrency control with suitable examples.**

In a Database Management System (DBMS), lock-based concurrency control (BCC**)** is a method used to manage how multiple transactions access the same data. This protocol ensures data consistency and integrity when multiple users interact with the database simultaneously.

This method uses locks to manage access to data, ensuring transactions don’t clash and everything runs smoothly when multiple transactions happen at the same time.

## What is a Lock?

A lock is a variable associated with a data item that indicates whether it is currently in use or available for other operations. Locks are essential for managing access to data during concurrent transactions. When one transaction is accessing or modifying a data item, a lock ensures that other transactions cannot interfere with it, maintaining data integrity and preventing conflicts. This process, known as locking, is a widely used method to ensure smooth and consistent operation in database systems.

## **Lock Based Protocols**

Lock-Based Protocols in DBMS ensure that a transaction cannot read or write data until it gets the necessary lock. Here’s how they work:

* These protocols prevent concurrency issues by allowing only one transaction to access a specific data item at a time.
* Locks help multiple transactions work together smoothly by managing access to the database items.
* Locking is a common method used to maintain the [serializability](https://www.geeksforgeeks.org/serializability-in-dbms/" \t "_blank) of transactions.
* A transaction must acquire a read lock or write lock on a data item before performing any read or write operations on it.

## Types of Lock

1. **Shared Lock (S):**Shared Lock is also known as Read-only lock. As the name suggests it can be shared between transactions because while holding this lock the transaction does not have the permission to update data on the data item. S-lock is requested using lock-S instruction.
2. **Exclusive Lock (X):** Data item can be both read as well as written. This is Exclusive and cannot be held simultaneously on the same data item. X-lock is requested using lock-X instruction.

## Rules of Locking

The basic rules for Locking are given below :

**Read Lock (or) Shared Lock(S)**

❖ If a Transaction has a Read lock on a data item, it can read the item but not update it.  
❖ If a transaction has a Read lock on the data item, other transaction can obtain Read Lock on the data item but no Write Locks.  
❖ So, the Read Lock is also called a Shared Lock.

**Write Lock (or) Exclusive Lock (X)**

❖ If a transaction has a write Lock on a data item, it can both read and update the data item.  
❖ If a transaction has a write Lock on the data item, then other transactions cannot obtain either a Read lock or write lock on the data item.  
❖ So, the Write Lock is also known as Exclusive Lock.

## **Concurrency Control Protocols**

Concurrency Control Protocols are the methods used to manage multiple transactions happening at the same time. They ensure that transactions are executed safely without interfering with each other, maintaining the accuracy and consistency of the database.

These protocols prevent issues like data conflicts, lost updates or inconsistent data by controlling how transactions access and modify data.

## Types of Lock-Based Protocols

### 1. Simplistic Lock Protocol

It is the simplest method for locking data during a transaction. Simple lock-based protocols enable all transactions to obtain a lock on the data before inserting, deleting, or updating it. It will unlock the data item once the transaction is completed.

**Example:**

Consider a database with a single data item X = 10.

**Transactions:**

* **T1**: Wants to read and update X.
* **T2**: Wants to read X.

#### Steps:

1. T1 requests an exclusive lock on X to update its value. The lock is granted.
   * T1 reads X = 10 and updates it to X = 20.
2. T2 requests a shared lock on X to read its value. Since T1 is holding an exclusive lock, T2 must wait.
3. T1 completes its operation and releases the lock.
4. T2 now gets the shared lock and reads the updated value X = 20.

This example shows how simplistic lock protocols handle concurrency but do not prevent problems like deadlocks or limits concurrency.

### 2. Pre-Claiming Lock Protocol

The Pre-Claiming Lock Protocol evaluates a transaction to identify all the data items that require locks. Before the transaction begins, it requests the database management system to grant locks on all necessary data elements. If all the requested locks are successfully acquired, the transaction proceeds. Once the transaction is completed, all locks are released. However, if any of the locks are unavailable, the transaction rolls back and waits until all required locks are granted before restarting.

#### Example:

Consider two transactions T1 and T2 and two data items, X and Y:

1. Transaction T1 declares that it needs:
   * A write lock on X.
   * A read lock on Y.

Since both locks are available, the system grants them. T1 starts execution:

* + It updates X.
  + It reads the value of Y.

1. While T1 is executing, Transaction T2 declares that it needs:
   * A read lock on X.

However, since T1 already holds a write lock on X, T2’s request is denied. T2 must wait until T1 completes its operations and releases the locks.

1. Once T1 finishes, it releases the locks on X and Y. The system now grants the read lock on X to T2, allowing it to proceed.

This method is simple but may lead to inefficiency in systems with a high number of transactions.

### 3. Two-phase locking (2PL)

A transaction is said to follow the Two-Phase Locking protocol if Locking and Unlocking can be done in two phases :

* **Growing 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.

For more detail refer the article [Two-phase locking (2PL).](https://www.geeksforgeeks.org/two-phase-locking-protocol/)

### 4. Strict Two-Phase Locking Protocol

Strict Two-Phase Locking requires that in addition to the 2-PL all Exclusive(X) locks held by the transaction be released until after the Transaction Commits.

For more details refer the article [Strict Two-Phase Locking Protocol](https://www.geeksforgeeks.org/categories-of-two-phase-locking-strict-rigorous-conservative/).

## Problem With Simple Locking

Consider the Partial Schedule:

| **S.No** | **T1** | **T2** |
| --- | --- | --- |
| 1 | lock-X(B) |  |
| 2 | read(B) |  |
| 3 | B:=B-50 |  |
| 4 | write(B) |  |
| 5 |  | lock-S(A) |
| 6 |  | read(A) |
| 7 |  | lock-S(B) |
| 8 | lock-X(A) |  |
| 9 | …… | …… |

### ****1. Deadlock****

In the given execution scenario, T1 holds an exclusive lock on B, while T2 holds a shared lock on A. At Statement 7, T2 requests a lock on B, and at Statement 8, T1 requests a lock on A. This situation creates a [deadlock](https://www.geeksforgeeks.org/deadlock-in-dbms/), as both transactions are waiting for resources held by the other, preventing either from proceeding with their execution.

### ****2. Starvation****

[Starvation](https://www.geeksforgeeks.org/starvation-in-dbms/) is also possible if concurrency control manager is badly designed. For example: A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item. This may be avoided if the concurrency control manager is properly designed.

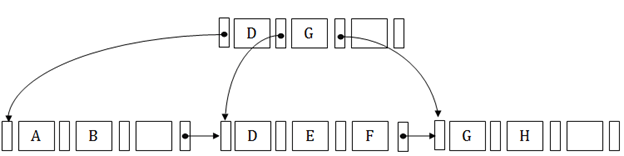
**10. Define B+ Trees. Explain operations of B+ Trees.**

# B+ Tree

* The B+ tree is a balanced binary search tree. It follows a multi-level index format.
* In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.
* In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.

## Structure of B+ Tree

* In the B+ tree, every leaf node is at equal distance from the root node. The B+ tree is of the order n where n is fixed for every B+ tree.
* It contains an internal node and leaf node.



## Internal node

* An internal node of the B+ tree can contain at least n/2 record pointers except the root node.
* At most, an internal node of the tree contains n pointers.

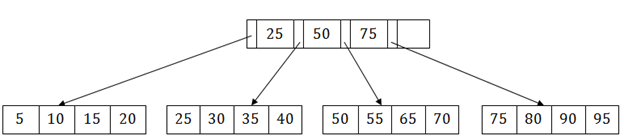
## Leaf node

* The leaf node of the B+ tree can contain at least n/2 record pointers and n/2 key values.
* At most, a leaf node contains n record pointer and n key values.
* Every leaf node of the B+ tree contains one block pointer P to point to next leaf node.

## Searching a record in B+ Tree

Suppose we have to search 55 in the below B+ tree structure. First, we will fetch for the intermediary node which will direct to the leaf node that can contain a record for 55.

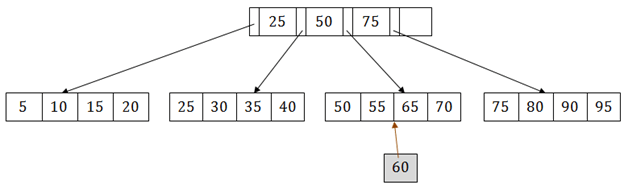
So, in the intermediary node, we will find a branch between 50 and 75 nodes. Then at the end, we will be redirected to the third leaf node. Here DBMS will perform a sequential search to find 55.



## B+ Tree Insertion

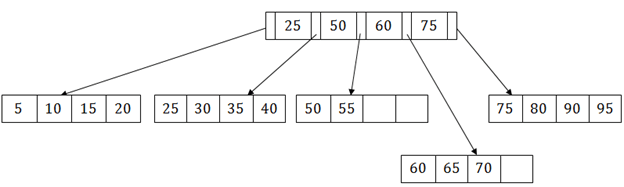
Suppose we want to insert a record 60 in the below structure. It will go to the 3rd leaf node after 55. It is a balanced tree, and a leaf node of this tree is already full, so we cannot insert 60 there.

In this case, we have to split the leaf node, so that it can be inserted into tree without affecting the fill factor, balance and order.



The 3rd leaf node has the values (50, 55, 60, 65, 70) and its current root node is 50. We will split the leaf node of the tree in the middle so that its balance is not altered. So we can group (50, 55) and (60, 65, 70) into 2 leaf nodes.

If these two has to be leaf nodes, the intermediate node cannot branch from 50. It should have 60 added to it, and then we can have pointers to a new leaf node.

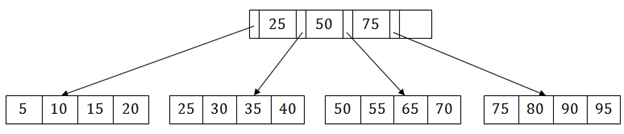


This is how we can insert an entry when there is overflow. In a normal scenario, it is very easy to find the node where it fits and then place it in that leaf node.

## B+ Tree Deletion

Suppose we want to delete 60 from the above example. In this case, we have to remove 60 from the intermediate node as well as from the 4th leaf node too. If we remove it from the intermediate node, then the tree will not satisfy the rule of the B+ tree. So we need to modify it to have a balanced tree.

After deleting node 60 from above B+ tree and re-arranging the nodes, it will show as follows:

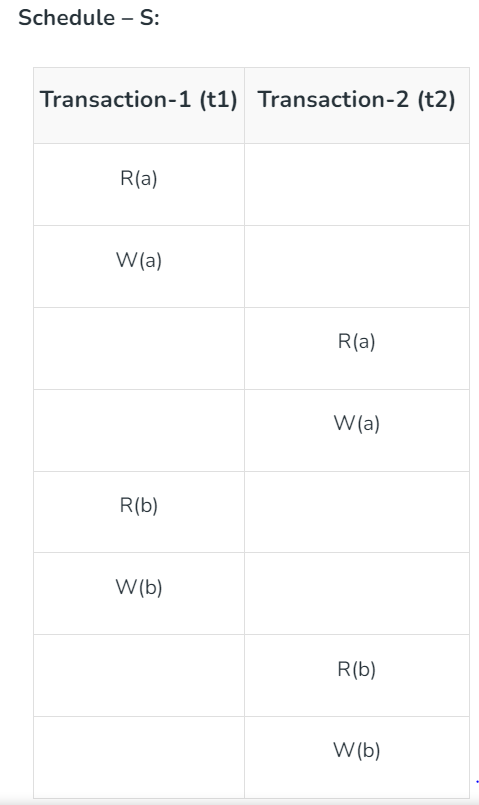
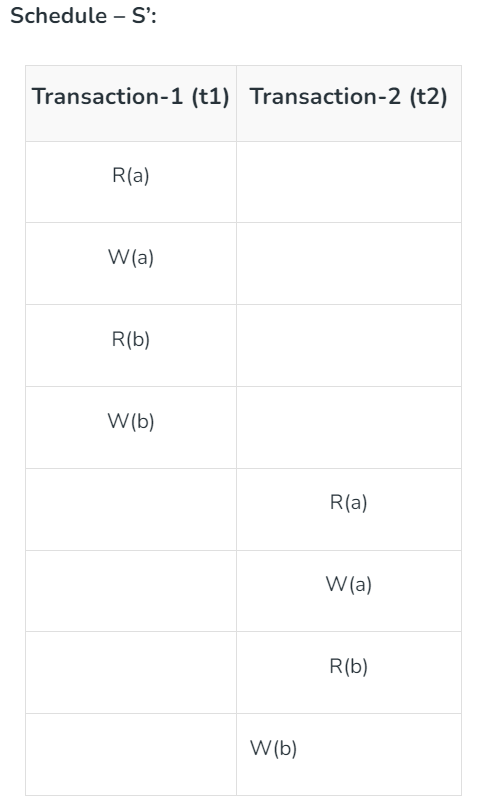


**Explain Serializability, Conflict Serializability and View Serializability.**

* Serializability of schedules ensures that a non-serial schedule is equivalent to a serial schedule.
* It helps in maintaining the transactions to execute simultaneously without interleaving one another. In simple words, serializability is a way to check if the execution of two or more transactions are maintaining the database consistency or not.
* Serializability in DBMS guarantees that the execution of multiple transactions in parallel does not produce any unexpected or incorrect results. This is accomplished by enforcing a set of rules that ensure that each transaction is executed as if it were the only transaction running in the system.

**Types of Serializability in DBMS**

There are mainly two types of serializability in DBMS:

* + View Serializability
  + Conflict Serializability
* **View Serializability**
* [View serializability](https://www.geeksforgeeks.org/view-serializability-in-dbms/) is a kind of operation in a serializable in which each transaction should provide some results, and these outcomes are the output of properly sequentially executing the data item. The view serializability, in contrast to conflict serialized, is concerned with avoiding database inconsistency. The view serializability feature of DBMS enables users to see databases in contradictory ways.
* To further understand view serializability in DBMS, we need to understand the schedules S1 and S2. The two transactions T1 and T2 should be used to establish these two schedules. Each schedule must follow the three transactions in order to retain the equivalent of the transaction. These three circumstances are listed below.
* The second requirement is that different read or write operations should not be used in either schedule. On the other hand, we say that two schedules are not similar if schedule S1 has two write operations whereas schedule S2 only has one. The number of the write operation must be the same in both schedules, however there is no issue if the number of the read operation is different.
* The second to last requirement is that there should not be a conflict between either timetable. execution order for a single data item. Assume, for instance, that schedule S1’s transaction is T1, and schedule S2’s transaction is T2. The data item A is written by both the transaction T1 and the transaction T2. The schedules are not equal in this instance. However, we referred to the schedule as equivalent to one another if it had the same number of all write operations in the data item.
* The first prerequisite is that the same kind of transaction appears on every schedule. This requirement means that the same kind of group of transactions cannot appear on both schedules S1 and S2. The schedules are not equal to one another if one schedule commits a transaction but it does not match the transaction of the other schedule.
* **What is view equivalency?**
* Schedules (S1 and S2) must satisfy these two requirements in order to be viewed as equivalent:
* The same piece of data must be read for the first time.For instance, if transaction t1 is reading “A” from the database in schedule S1, then t1 must also read A in schedule S2.
* The same piece of data must be used for the final write. As an illustration, if transaction t1 updated A last in S1, it should also conduct final write in S2.
* The middle sequence need to follow suit. As an illustration, if in S1 t1 is reading A, and t2 updates A, then in S2 t1 should read A, and t2 should update A.
* View Serializability refers to the process of determining whether a schedule’s views are equivalent.
* **Example**
* We have a schedule “S” with two concurrently running[transactions](https://www.geeksforgeeks.org/sql-transactions/),“t1” and “t2.”
  + 
* By switching between both transactions’ mid-read-write operations, let’s create ts view equivalent schedule (S’).
  + 
  + **Conflict serializability**
* [Conflict serializability](https://www.geeksforgeeks.org/conflict-serializability-in-dbms/) refers to a subset of serializability that focuses on maintaining the consistency of a database while ensuring that identical data items are executed in an order. In a DBMS each transaction has a value and all the transactions, in the database rely on this uniqueness. This uniqueness ensures that no two operations with the conflict value can occur simultaneously.
* For example lets consider an order table and a customer table as two instances. Each order is associated with one customer even though a single client may place orders. However there are restrictions for achieving conflict serializability in the database. Here are a few of them.
* Different transactions should be used for the two procedures.
* The identical data item should be present in both transactions.
  + Between the two operations, there should be at least one write operation.
* **Example**
* Three transactions—t1, t2, and t3—are active on a schedule “S” at once. Let’s create a graph of precedence.



**Write notes on 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.

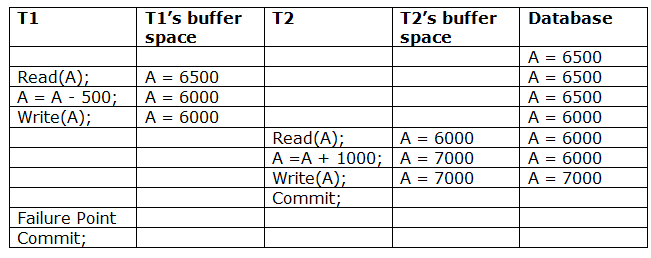
**Undo-redo logging:**This level of recoverability provides both undo and redo capabilities, ensuring that the system can recover to a consistent state regardless of whether a transaction has been committed or not.

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.

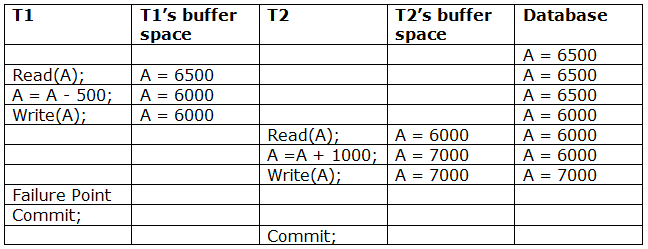
# Recoverability of Schedule

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



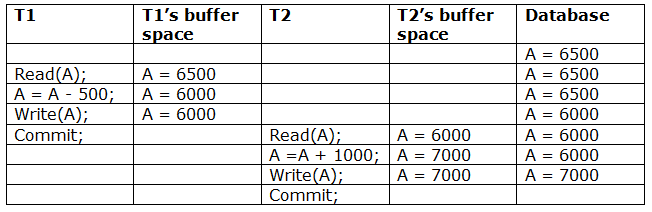
The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1, but T2 can't be rollback because it already committed. So this type of schedule is known as irrecoverable schedule.

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



The above table 2 shows a schedule with two transactions. Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, we have to rollback T1. T2 should be rollback because T2 has read the value written by T1. As it has not committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with cascade rollback.

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



The above Table 3 shows a schedule with two transactions. Transaction T1 reads and write A and commits, and that value is read and written by T2. So this is a cascade less recoverable schedule.

### The capabilities of recoverability in a DBMS encompass:

**Atomicity:**Transactions in a DBMS are designed to be atomic, which means that they either entire absolutely or are rolled back to their unique nation in case of a failure. This guarantees that the database is usually in a consistent nation.

**Durability:**Once a transaction is dedicated, its changes are permanently stored to the database, even in the occasion of a failure. This ensures that the database may be restored to its closing consistent kingdom after a failure.

**Logging:**A DBMS keeps a log of all transactions to make sure recoverability. The log consists of data about all adjustments made to the database, as well as the transactions that made those changes. In the event of a failure, the log may be used to repair the database to a regular state.

**Checkpointing:**A checkpoint is a point in time in which the DBMS records the country of the database and logs it. This allows lessen the quantity of time required for recovery in case of a failure, as handiest the transactions since the last checkpoint need to be rolled back or replayed.

**Recovery manager:** A restoration supervisor is a part of a DBMS that is accountable for restoring the database to a constant state after a failure. The healing supervisor uses the log and checkpoints to determine which transactions want to be rolled returned or replayed to repair the database.

**Media recuperation:**Media recovery refers back to the capability of a DBMS to recover from a failure that affects the garage media, such as a hard disk crash. This includes restoring the database from a backup and making use of the log to deliver it up to date

**Explain about Failure Classification.**

* A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, a fire in the machine room, even damage. In any failure, information may be lost.
* Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions.
* There are various types of failure that may occur in a system:

1. Transaction failure

2. System crash

3. Disk failure

1. Transaction failure

* The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further.
* If a few transaction or process is hurt, then this is called as transaction failure.
* Reasons for a transaction failure could be –

Logical errors: If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.

Syntax error: It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. For example, The system aborts an active transaction, in case of deadlock or resource unavailability

2.System Crash: System failure can occur due to power failure or other hardware or software failure.

Example: Operating system error.

Fail-stop assumption: In the system crash, non-volatile storage is assumed not to be corrupted..

3. Disk Failure:

* It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.
* Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

**Explain about Hash based indexing (Static Hashing and Dynamic Hashing).**

Hashing in DBMS is a technique to quickly locate a data record in a database irrespective of the size of the database. For larger databases containing thousands and millions of records, the indexing data structure technique becomes very inefficient because searching a specific record through indexing will consume more time. This doesn’t align with the goals of DBMS, especially when performance and date retrieval time are minimized. So, to counter this problem hashing technique is used. In this article, we will learn about various hashing techniques.

**What is Hashing?**

The hashing technique utilizes an auxiliary hash table to store the data records using a hash function. There are 2 key components in hashing:

* **Hash Table:**A hash table is an array or data structure and its size is determined by the total volume of data records present in the database. Each memory location in a hash table is called a ‘***bucket***‘ or hash indice and stores a data record’s exact location and can be accessed through a hash function.
* **Bucket:**A bucket is a memory location (index) in the hash table that stores the data record. These buckets generally store a disk block which further stores multiple records. It is also known as the hash index.
* **Hash Function:**A hash function is a mathematical equation or algorithm that takes one data record’s primary key as input and computes the hash index as output.

**Hash Function**

A hash function is a mathematical algorithm that computes the index or the location where the current data record is to be stored in the hash table so that it can be accessed efficiently later. This hash function is the most crucial component that determines the speed of fetching data.

### Working of Hash Function

The hash function generates a hash index through the primary key of the data record.

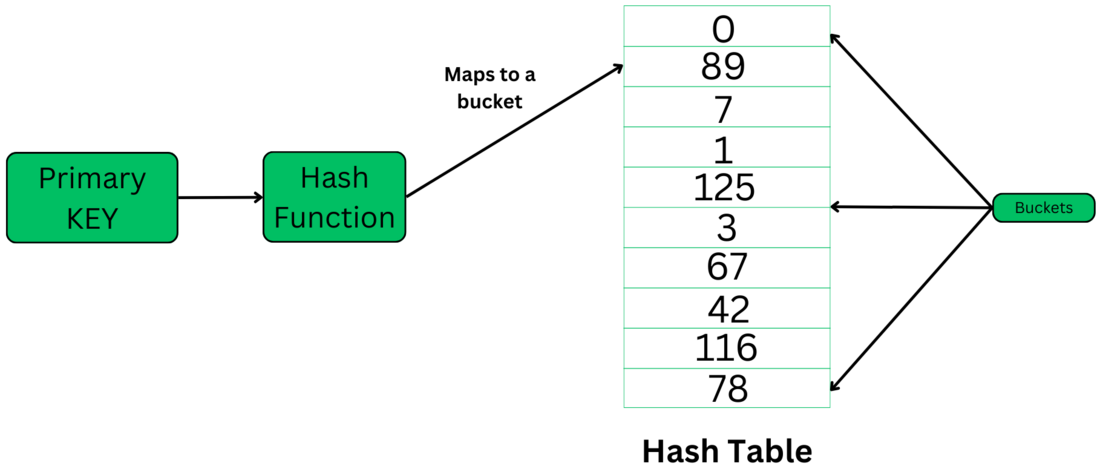
**Now, there are 2 possibilities:**

1. The hash index generated isn’t already occupied by any other value. So, the address of the data record will be stored here.

2. The hash index generated is already occupied by some other value. This is called collision so to counter this, a collision resolution technique will be applied.

3. Now whenever we query a specific record, the hash function will be applied and returns the data record comparatively faster than indexing because we can directly reach the exact location of the data record through the hash function rather than searching through indices one by one.

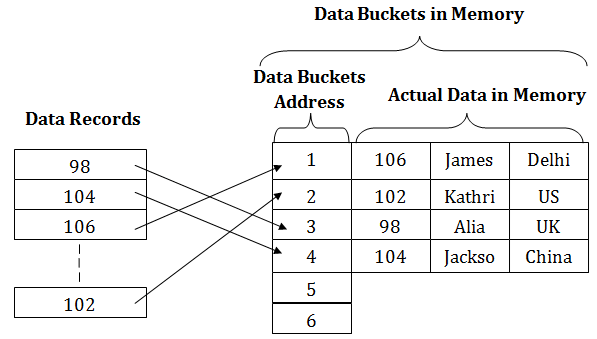
**Example:**



# Static Hashing

In static hashing, the resultant data bucket address will always be the same. That means if we generate an address for EMP\_ID =103 using the hash function mod (5) then it will always result in same bucket address 3. Here, there will be no change in the bucket address.

Hence in this static hashing, the number of data buckets in memory remains constant throughout. In this example, we will have five data buckets in the memory used to store the data.



## Operations of Static Hashing

* **Searching a record**

When a record needs to be searched, then the same hash function retrieves the address of the bucket where the data is stored.

* **Insert a Record**

When a new record is inserted into the table, then we will generate an address for a new record based on the hash key and record is stored in that location.

* **Delete a Record**

To delete a record, we will first fetch the record which is supposed to be deleted. Then we will delete the records for that address in memory.

* **Update a Record**

To update a record, we will first search it using a hash function, and then the data record is updated.

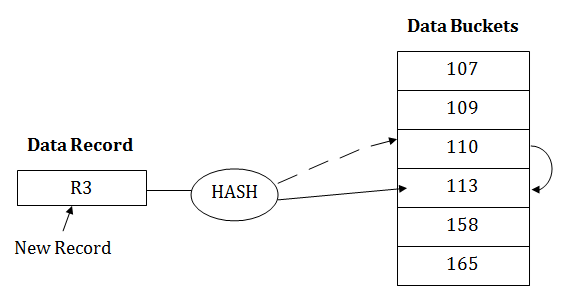
If we want to insert some new record into the file but the address of a data bucket generated by the hash function is not empty, or data already exists in that address. This situation in the static hashing is known as **bucket overflow**. This is a critical situation in this method.

To overcome this situation, there are various methods. Some commonly used methods are as follows:

## 1. Open Hashing

When a hash function generates an address at which data is already stored, then the next bucket will be allocated to it. This mechanism is called as **Linear Probing**.

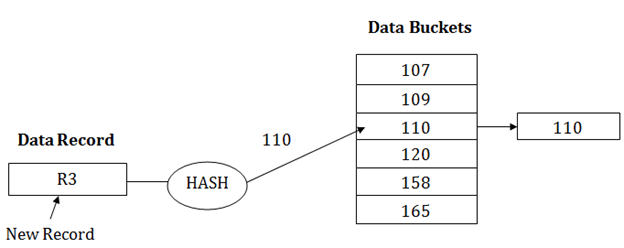
**For example:** suppose R3 is a new address which needs to be inserted, the hash function generates address as 112 for R3. But the generated address is already full. So the system searches next available data bucket, 113 and assigns R3 to it.



## 2. Close Hashing

When buckets are full, then a new data bucket is allocated for the same hash result and is linked after the previous one. This mechanism is known as **Overflow chaining**.

**For example:** Suppose R3 is a new address which needs to be inserted into the table, the hash function generates address as 110 for it. But this bucket is full to store the new data. In this case, a new bucket is inserted at the end of 110 buckets and is linked to it.



# Dynamic Hashing

ADVERTISEMENT

* The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
* In this method, data buckets grow or shrink as the records increases or decreases. This method is also known as Extendable hashing method.
* This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

## How to search a key

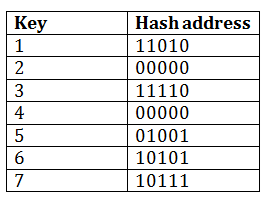
* First, calculate the hash address of the key.
* Check how many bits are used in the directory, and these bits are called as i.
* Take the least significant i bits of the hash address. This gives an index of the directory.
* Now using the index, go to the directory and find bucket address where the record might be.

## How to insert a new record

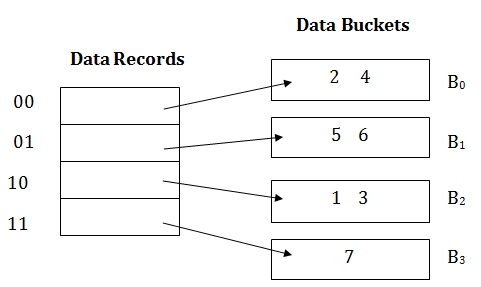
* Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
* If there is still space in that bucket, then place the record in it.
* If the bucket is full, then we will split the bucket and redistribute the records.

## For example:

Consider the following grouping of keys into buckets, depending on the prefix of their hash address:

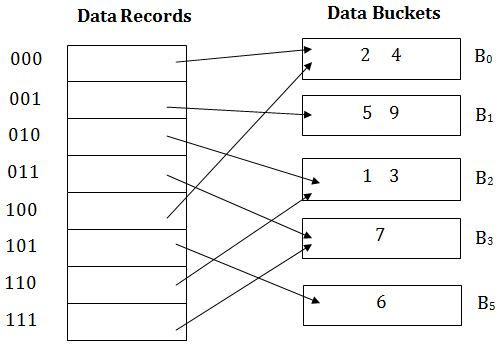


The last two bits of 2 and 4 are 00. So it will go into bucket B0. The last two bits of 5 and 6 are 01, so it will go into bucket B1. The last two bits of 1 and 3 are 10, so it will go into bucket B2. The last two bits of 7 are 11, so it will go into B3.



## Insert key 9 with hash address 10001 into the above structure:

* Since key 9 has hash address 10001, it must go into the first bucket. But bucket B1 is full, so it will get split.
* The splitting will separate 5, 9 from 6 since last three bits of 5, 9 are 001, so it will go into bucket B1, and the last three bits of 6 are 101, so it will go into bucket B5.
* Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00.
* Keys 1 and 3 are still in B2. The record in B2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.
* Key 7 are still in B3. The record in B3 pointed by the 111 and 011 entry because last two bits of both the entry are 11.



## Advantages of dynamic hashing

* In this method, the performance does not decrease as the data grows in the system. It simply increases the size of memory to accommodate the data.
* In this method, memory is well utilized as it grows and shrinks with the data. There will not be any unused memory lying.
* This method is good for the dynamic database where data grows and shrinks frequently.

## Disadvantages of dynamic hashing

* In this method, if the data size increases then the bucket size is also increased. These addresses of data will be maintained in the bucket address table. This is because the data address will keep changing as buckets grow and shrink. If there is a huge increase in data, maintaining the bucket address table becomes tedious.
* In this case, the bucket overflow situation will also occur. But it might take little time to reach this situation than static hashing.

**Explain about Optimistic (Validation) Concurrency protocol.**

**Validation Based Protocol** is also called Optimistic Concurrency Control Technique. This protocol is used in DBMS (Database Management System) for avoiding concurrency in transactions. It is called optimistic because of the assumption it makes, i.e. very less interference occurs, therefore, there is no need for checking while the transaction is executed.

In this technique, no checking is done while the transaction is been executed. Until the transaction end is reached updates in the transaction are not applied directly to the database. All updates are applied to local copies of data items kept for the transaction. At the end of transaction execution, while execution of the transaction, a **validation phase** checks whether any of transaction updates violate serializability. If there is no violation of serializability the transaction is committed and the database is updated; or else, the transaction is updated and then restarted.

Optimistic Concurrency Control is a three-phase protocol. The three phases for validation based protocol:

1. **Read Phase:**   
   Values of committed data items from the database can be read by a transaction. Updates are only applied to local data versions.
2. **Validation Phase:**   
   Checking is performed to make sure that there is no violation of serializability when the transaction updates are applied to the database.
3. **Write Phase:**   
   On the success of the validation phase, the transaction updates are applied to the database, otherwise, the updates are discarded and the transaction is slowed down.

The idea behind optimistic concurrency is to do all the checks at once; hence transaction execution proceeds with a minimum of overhead until the validation phase is reached. If there is not much interference among transactions most of them will have successful validation, otherwise, results will be discarded and restarted later. These circumstances are not much favourable for optimization techniques, since, the assumption of less interference is not satisfied.

Validation based protocol is useful for rare conflicts. Since only local copies of data are included in rollbacks, cascading rollbacks are avoided. This method is not favourable for longer transactions because they are more likely to have conflicts and might be repeatedly rolled back due to conflicts with short transactions.

In order to perform the Validation test, each transaction should go through the various phases as described above. Then, we must know about the following three time-stamps that we assigned to transaction Ti, to check its validity:

**1. Start(Ti):** It is the time when Ti started its execution.

**2. Validation(Ti):** It is the time when Ti just ﬁnished its read phase and begin its validation phase.

**3. Finish(Ti):** the time when Ti end it’s all writing operations in the database under write-phase.

Two more terms that we need to know are:

**1. Write\_set:**of a transaction contains all the write operations that Ti performs.

**2. Read\_set:**of a transaction contains all the read operations that Ti performs.

In the Validation phase for transaction Ti the protocol inspect that Ti doesn’t overlap or intervene with any other transactions currently in their validation phase or in committed. The validation phase for Ti checks that for all transaction Tj one of the following below conditions must hold to being validated or pass validation phase:

**1.** **Finish(Tj)<Starts(Ti)**, since Tj finishes its execution means completes its write-phase before Ti started its execution(read-phase). Then the serializability indeed maintained.

**2.** Ti begins its write phase after Tj completes its write phase, and the read\_set of Ti should be disjoint with write\_set of Tj.

**3.** Tj completes its read phase before Ticompletes its read phase and both read\_set and write\_set of Ti are disjoint with the write\_set of Tj.

**Ex: Here two Transactions Ti and Tj are given, since TS(Tj)<TS(Ti)**so the validation phase succeeds in the Schedule-A. It’s noteworthy that the final write operations to the database are performed only after the validation of both Ti and Tj. Since Ti reads the old values of**x(12)**and**y(15)**while **print(x+y)**operation unless final write operation take place.

**Schedule-A**

| **Tj** | **Ti** |
| --- | --- |
| **r(x) // x=12** |  |
|  | **r(x)** |
|  | **x=x-10**  **r(y) //y=15** |
|  | **y=y+10**  **r(x)** |
| **<validate>**  **print(x+y)** |  |
|  | **<validate>** |
|  | **w(x)**  **w(y)** |

**Schedule-A is a validated schedule**

**Advantages:**

**1. Avoid Cascading-rollbacks:**This validation based scheme avoid cascading rollbacks since the final write operations to the database are performed only after the transaction passes the validation phase. If the transaction fails then no updation operation is performed in the database. So no dirty read will happen hence possibilities cascading-rollback would be null.

**2. Avoid deadlock:**Since a strict time-stamping based technique is used to maintain the specific order of transactions. Hence deadlock isn’t possible in this scheme.

**Disadvantages:**

**1. Starvation:**There might be a possibility of starvation for long-term transactions, due to a sequence of conﬂicting short-term transactions that cause the repeated sequence of restarts of the long-term transactions so on and so forth. To avoid starvation, conﬂicting transactions must be temporarily blocked for some time, to let the long-term transactions to ﬁnish.

**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

* The highest isolation level.
* Guarantees full serializability and ensures complete isolation of transaction operations.

## 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

* 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

* 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 proper isolation level or concurrency control mechanism to use depends on the specific requirements of a system and its workload. Some systems may prioritize high throughput and can tolerate lower isolation levels, while others might require strict consistency and higher isolation.

| **Isolation level** | **Dirty Read** | **Unrepetable Read** |
| --- | --- | --- |
| Serializable | NO | NO |
| Repeatable Read | NO | NO |
| Read Committed | NO | Maybe |
| Read Uncommitted | Maybe | Maybe |

# 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.

* **Shared Lock (S-lock):** Allows a transaction to read an item but not write to it.
* **Exclusive Lock (X-lock):** Allows a transaction to read and write an item. No other transaction can read or write until the lock is released.
* **Two-phase Locking (2PL):** This protocol ensures that a transaction acquires all the locks before it releases any. This results in a growing phase (acquiring locks and not releasing any) and a shrinking phase (releasing locks and not acquiring any).

# 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.