DBMS MODULE 2 NOTES

***Database Design & Normalization:** Lossless Join and Dependency Preserving Decomposition, MVD and 4th Normal Form, JD and 5th Normal Form, Inclusion Dependence.

Decomposition: The process of breaking up of a relation into smaller sub relations is called Decomposition. Decomposition is required in DBMS to convert a relation into a specific normal form which further reduces redundancy, anomalies, and inconsistency in the relation.

There are mainly two types of decompositions in DBMS-

1. Lossless Decomposition
2. Lossy Decomposition

Lossless	Lossy
The decompositions R1, R2, R2…Rn for a relation schema R are said to be Lossless if there natural join results the original relation R.	The decompositions R1, R2, R2…Rn for a relation schema R are said to be Lossy if there natural join results into addition of extraneous tuples with the original relation R.
Formally, Let R be a relation and R1, R2, R3 … Rn be it's decomposition, the decomposition is lossless if – $R1 \bowtie R2 \bowtie R3.... \bowtie Rn = R$	Formally, Let R be a relation and R1, R2, R3 … Rn be its decomposition, the decomposition is lossy if – $R \subset R1 \bowtie R2 \bowtie R3.... \bowtie Rn$
There is no loss of information as the relation obtained after natural join of decompositions is equivalent to original relation. Thus, it is also referred to as non-additive join decomposition	There is loss of information as extraneous tuples are added into the relation after natural join of decompositions. Thus, it is also referred to as careless decomposition.
The common attribute of the sub relations is a super key/candidate key of any one of the relation.	The common attribute of the sub relation is not a super key/candidate key of any of the sub relation.

1 NF	2 NF	3 NF	BCNF
Is always LOSSLESS	**Is always LOSSLESS**	**Is always LOSSLESS**	**May be LOSSLESS or LOSSY**
Is always DEPENDENCY PRESERVING	**Is always DEPENDENCY PRESERVING**	**Is always DEPENDENCY PRESERVING**	**May be DEPENDENCY PRESERVING or not be DEPENDENCY**

			PRESERVING

Imp: The 1NF, 2NF, and 3NF are valid for dependency preserving decomposition.

Dependency Preserving Decomposition is a technique used in Database Management System (DBMS) to decompose a relation into smaller relations while preserving the functional dependencies between the attributes. The goal is to improve the efficiency of the database by reducing redundancy and improving query performance.

In this technique, the original relation is decomposed into smaller relations in such a way that the resulting relations preserve the **functional dependencies** of the original relation. This is important because if the decomposition results in losing any of the **original functional dependencies**, it can lead to data inconsistencies and anomalies.

Decomposition is Dependency Preserving if F1 ∪ F2 ∪ F3 ∪… U Fn = F

Decomposition is not Dependency Preserving if F1 ∪ F2 ∪ F3 ∪… U Fn ⊂ F

EXAMPLE 1:

Given Relation: R(X, Y, Z)

Functional Dependency: X -> Y, Y -> Z, Z -> X

Decompositions: R_1 (X, Y) & R_2 (Y, Z)

Check whether it is lossy or lossless also dependency is preserving or not.

SOLUTION:

For Lossy: Common attribute (Y i.e., candidate key) is present in both R1 and R2. Therefore, Relation is not lossy.

For Lossless:

- All the attributes of R should be present in R1 & R2.

i.e., R1 ∪ R2 = R

(X, Y) ∪ (Y, Z) = (X, Y, Z)

- Common attributes between R1 and R2 should not be null.

R1 ∩ R2 ≠ Ø

(X, Y) ∩ (Y, Z) = (Y) {Y is a distinct value or candidate key}

Hence, Relation is Lossless.

For Dependency Preserving:

Step 1: Write down all the non trivial dependency present in the child table R1 and R2

R1 (X, Y)	**R2 (Y, Z)**
FD: X -> Y **Y -> X**	**FD: Y -> Z** **Z -> Y**

Step 2: Check the dependency hold true or not:-

If {Dependency is directly present (in the given question) then True}

Else {Take the closure of the determinant w.r.t functional dependency given in the question and check the possibility. Determinant present on LHS.}

Check the dependency which is directly present (in the given question) : True

R1 (X, Y)	**R2 (Y, Z)**
FD: X -> Y (directly present, True)	**FD: Y -> Z** (directly present, True)

Y -> X (not present directly)	**Z -> Y** (not present directly)
Take the closure of determinant Y^+ = (Y, Z, X) Here, Z is not the part of R1, so, exclude Z Hence, we proved that Y can determine X indirectly. (True)	Take the closure of determinant Z^+ = (Z, X, Y) Here, X is not the part of R2, so, exclude X Hence, we proved that Z can determine Y indirectly. (True)

Step 3: Take the union of all possible functional dependencies.

X -> Y

Y -> X

Y -> Z

Z -> Y

Step 4: Check all the original dependencies are **preserved** or not, w.r.t. to possible functional dependencies.

If {Dependency is directly preserved then True}

Else {Take the closure of the determinant w.r.t. derived functional dependencies}

Original Dependencies (in the given question)	**Derived Dependencies**
X -> Y Y -> Z Z -> X	X -> Y (Directly Preserved) Y -> X Y -> Z (Directly Preserved) Z -> Y

Check for Z -> X which is not directly present.

Therefore, Take the closure of Determinant Z w.r.t. to **derived** Functional Dependencies

Z -> X

$Z^+ = \{Z, Y, X\}$

Here, Z determining X indirectly is hence, True

Therefore, Z -> X is also preserved.

Thus, Decomposition is Dependency Preserving.

EXAMPLE 2:

GIVEN RELATION: R (A, B, C, D)

Functional Dependencies: AB -> CD, D-> A

Find Candidate Key, Check for 3NF, BCNF, Lossy or Lossless, Dependency Preserving or Not?

SOLUTION:

For Candidate Keys:

AB -> CD

D -> A

Variables present at Right Hand Side of the given functional dependencies are A, C, D

B is not present at RHS

Therefore, find the closure of B^+

$B^+ = \{B\}$ *not a candidate key*

Assume

$AB^+ = \{A, B, C, D\}$ *candidate key*

$CB^+ = \{C, B\}$ *not a candidate key*

$DB^+ = \{D, B, A, C\}$ *candidate key*

Candidate Keys are: AB^+, DB^+

Prime Attributes: {A, B, D}

Non Prime Attributes: {C}

For 3NF: Relation is in 3NF if

LHS of functional dependency must be a Candidate Key/Super Key.

OR

RHS of functional dependency must have Prime Attributes.

AB -> CD *here LHS is a Candidate key*

D -> A *here RHS is a prime attribute*

Therefore, Relation R (A, B, C, D) is in 3NF

FOR BCNF: Relation is in BCNF if

LHS of functional dependency must be a Candidate Key/Super Key.

AB -> CD *here LHS is a Candidate key*

D -> A *here LHS is not a Candidate key*

Therefore, Relation R (A, B, C, D) is not in BCNF

Note: *If the relation is not in BCNF convert it into BCNF.*

We use Decomposition to normalize a table/relation.

Decompose the Relation R(A, B, C, D) into R1 and R2

AB -> CD *here LHS is a Candidate key*

D -> A *here LHS is not a Candidate key*

Find the closure of **D**

$D^+ = \{DA\}$

Hence, Decompositions are: R1 (D, A) and R2 (B, C, D)

D is common in both R1 and R2 so, D is a Candidate Key.

Since, D is our candidate key

Now, again check D -> A *here LHS is a Candidate key*

Therefore, Now the Relation is in BCNF.

BCNF is Lossy or Lossless: R1 (D, A) and R2 (B, C, D)

For Lossy: D is common here i.e., Candidate Key. So, the relation is not Lossy

For Lossless:

- All the attributes of R should be present in R1 & R2.

i.e., $R1 \cup R2 = R$

$(D, A) \cup (B, C, D) = (A, B, C, D)$

- Common attributes between R1 and R2 should not be null.

$R1 \cap R2 \neq \emptyset$

$(D, A) \cap (B, C, D) = (D)$ {D is a distinct value or candidate key}

Hence, Relation is Lossless.

For Dependency Preserving in 3NF: Always Preserving

For Dependency Preserving in BCNF:

Check the original dependencies (given in the question)

AB -> CD

D -> A

R1 (D, A)	**R2 (B, C, D)**
{D -> A} *dependency preserved*	Closure B+ = {B} C+ = {C} D+ = {D} BC+ = {BC} BD+ = {BDAC} *candidate key* *Hence, for* **R2 (B, C, D)** {BD -> C} *Here, BD is a candidate key* AB -> CD *dependency not preserved*

Hence, Decomposition R1 (D,A) and R2 (B, C, D) is not dependency preserving.

Hence, BCNF is not Dependency Preserving

4NF WITH MULTI VALUED DEPENDENCY (MVD)

4NF:-

A relation "R" is said to be 4NF if & only if the following conditions are satisfied:

1) "R" is already in 3NF or BCNF

2) If it not contains MVD

MVD (→→) :-

1) There must be three or more attributes.

2) Attribute must be independent of each other.

EXAMPLE:

Consider a following table **Student** (name, operating system, language)

Name	Operating System	Language
Aman	Windows Apple	English Hindi
Mohan	Linux	English Spanish

Normalize the table. Is this table is in 4NF? If no decompose it into 4NF.

Solution: Normalize the table (Each cell holds single value)

Name	Operating System	Language
Aman	Windows	English
Aman	Windows	Hindi
Aman	Apple	English
Aman	Apple	Hindi
Mohan	Linux	English
Mohan	Linux	Spanish

Here, we have 3 attributes in the above table. Therefore MVD is there.

If the table contains MVD it means it is not in 4NF.

Now,

Decompose the table to normalize it.

Name →→ Operating System (R1) *name depends on os*

Name →→ Language (R2) *name depends on language*

R1 (Name, Operating System)		R2 (Name, Language)	
Name	Operating System	Name	Language
Aman	Windows	Aman	English
Aman	Apple	Aman	Hindi

Mohan	Linux	Mohan	English
		Mohan	Spanish
No MVD in table R1 because R1 Contains two attributes.		No MVD in table R2 because R2 Contains two attributes.	

Hence, Relation is in 4NF it it has no MVD.

5NF WITH JOIN DEPENDENCY (JD)

5NF:-

A relation "R" is said to be 5NF if & only if the following conditions are satisfied simultaneously:

1) A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
2) 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
3) 5NF is also known as Project-join normal form (PJ/NF).

JD:-

Let "R" be a relation schema & R1, R2,……,Rn be the decomposition of R, R is said to satisfy the join dependency if and only if

$$\Pi_{(R1)}(R) \bowtie \Pi_{(R2)}(R) \bowtie \Pi_{(R3)}(R) \bowtie \ldots\ldots\ldots \Pi_{(Rn)}(R) \bowtie = R$$

EXAMPLE:

Consider a relation involving suppliers, parts, and projects:

Initial Table (Supplier, Parts, Projects)

Supplier	Part	Project
S1	P1	J1
S1	P2	J1
S1	P1	J2

S2	P2	J2

Given the above constraints, the following join dependencies exist on the table:

{Supplier, Part} ⟶ Supplier Part Project

{Supplier, Project} ⟶ Supplier Part Project

{Part, Project} ⟶ Supplier Part Project

To decompose the relation into 5NF:

Supplier	**Part**
S1	P1
S1	P2
S2	P2

Supplier	**Project**
S1	J1
S1	J2
S2	J2

Part	**Project**
P1	J1
P2	J1
P1	J2
P2	J2

Now, these decomposed tables eliminate the redundancy caused by the specific constraints and join dependencies of the original relation.

When you take the natural join of these tables, you will get back the original table.

Reaching 5NF can lead to an increased number of tables, which can complicate queries and database operations. Thus, achieving 5NF should be a conscious decision made based on the specific requirements and constraints of a given application.

INCLUSION DEPENDENCE

Imagine you have two lists, one with names of your friends and another with their phone numbers. Each friend's name is matched with their phone number. Now, if every friend's name in the first list is also found in the second list along with their phone number, we can say there is an "inclusion dependence" between the two lists.

So, in simple terms, inclusion dependence means that if something is in one list, it must also be in another list. It's like saying, "If it's on the friend's name list, it has to be on the phone number list too."

In a database, it's similar. If you have information in one table, and there's an inclusion dependence with another table, it means that every piece of information in the first table must also be in the second table.

A statement in which some columns of any relation are contained in other columns is known as an Inclusion Dependency. Inclusion dependencies, like functional dependencies, represent one-to-many relationships. However, inclusion dependencies are more commonly used to represent relationships between relations.

Let's name the relations R as teacher and S as student, so take the attribute as teacher_id, so we can write:

teacher.teacher_id --> student.teacher_id

Teacher:

teacher_id (primary key)	name	department
1	Ram Kumar	DBMS

Student:

student	name	teacher_id (foreign key)	age

student	name	teacher_id (foreign key)	age
1	Rahul Singh	1	18

teacher_id will be the primary key for teacher table and will be foreign key for the student table, attributes of the teacher table will be available in the student table.

So this **foreign key** concept makes the **inclusion dependency** possible.

File Organization: Indexing, Structure of Index Files and Types, Dense and Sparse Indexing

*****INDEXING:** Indexing is a very useful technique that helps in optimizing the search time in database queries. The table of database indexing consists of a search key and pointer.

The index is a type of data structure. It is used to locate and access the data in a database table quickly.

There are four types of indexing:

- **Primary Indexing:**
  - Indexing done on primary key or any super key field.
  - Data must be ordered on index field.
  - It is always Sparse Index.
- **Clustering Indexing:**
  - Indexing done on non key field.
  - Data must be ordered on index field.
  - It is always Sparse Index.
- **Secondary Key Indexing:**
  - Indexing done on primary key or any super key field.
  - Data must be unordered on index field.

- It can be Sparse or Dense Index.

- **Secondary Non Key Indexing:**
  - Indexing done on non key field.
  - Data must be unordered on index field.
  - It is always Dense Index.

SUMMARY:

INDEXING	ORDERING	KEY OR NON KEY	SPARSE/DENSE
Primary Indexing	Ordered	Key	Sparse
Clustering Indexing	Ordered	Non-Key	Sparse
Secondary Key Indexing	UnOrdered	Key	Sparse or Dense
Secondary Non Key Indexing	UnOrdered	Non-Key	Dense

INDEX STRUCTURE:

Indexes can be created using some database columns.

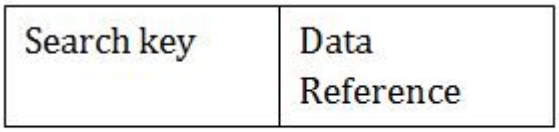


Fig: Structure of Index

The first column of the database is the search key that contains a copy of the primary key or candidate key of the table. The values of the primary key are stored in sorted order so that the corresponding data can be accessed easily.

The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.

Dense index VS Sparse index	Dense Index	Sparse Index
Index size	Larger	Smaller
Records in data file	Need not be clustered	Need to be Clustered
Time to locate data	Less	More
Computing time in RAM	Less	More
Overhead for insertions and deletions	More	Less
Data pointers pointing to	Each record in the data file	Fewer records in the data file
Efficiency	Less computationally efficient than sparse layers	More computationally efficient than dense layers
Ordering	The dense index can be built on **order** as well as **unordered** fields of the database files.	The sparse index can be built only on the **ordered** field of the database file.

Multi Level Indexing - B Tree & B^+ Tree

- **Block Pointer:** Points the block
- **Record Pointer:** Points to the record we are searching for
- **Root:** A root can have children between 2 & P {*here, P is called the order of tree*}
- **Order of Tree:** Maximum number of children a node can have
- **Internal Node:** Also called intermediate nodes
- **External Node/ Leaf Node:** Nodes with no child called leaf nodes

	Root	Intermediate Node
Max	**P**	**P**
Min	**2**	**P/2**

Keys	Min no. of keys	Max no. of keys
Root	**2-1**	**P-1**
Internal Node	**P/2 - 1**	**P-1**
Leaf Node	**P/2 - 1**	**P-1**

Example: Consider a B^+ Tree in which the maximum number of keys in a node is 5. What is the order of tree, minimum number of keys in root node, internal node and leaf node?

Solution: Given: Maximum number of keys = 5

Maximum number of keys = P-1 {*refer table*}

Therefore, P-1 = 5

P = 5 + 1 = 6

Order of Tree, P = 6

Minimum number of keys in root node = 2 - 1 = 1

Minimum number of keys in internal node = P/2 - 1 = 6/2 - 1 = 2

Minimum number of keys in leaf node = P/2 - 1 = 6/2 - 1 = 2

Example: The order of a leaf node in a B^+ tree is the maximum number of (value, data record pointer) pairs it can hold. Given that the block size is 1K bytes, data record pointer is 7 bytes long, the value field is 9 bytes long and a block pointer is 6 bytes long, what is the order of the leaf node, maximum number of keys?

Solution: Given: Block Size = 1K Bytes = 1024 Bytes

Record Pointer = 7 Bytes

Key Size = 9 Bytes

Block Pointer = 6 Bytes

Order, P = ?

Maximum number of keys = ?

Formula: P[(Key Size + Record Pointer Size) + Block Pointer Size] ≤ Block Size

P[(9 + 7) + 6] ≤ 1024 Bytes

16P + 6 ≤ 1024

16P = 1024 - 6

16P = 1018

P = 1018/16

P = 63

Therefore, order of tree, P = 63

Maximum Number of Keys = P -1 = 63 - 1 = 62

Example: A data file consisting of 1,50,000 student-records is stored on a hard disk with block size of 4096 bytes. The data file is sorted on the primary key RollNo. The size of a record pointer for this disk is 7 bytes. Each student-record has a candidate key attribute called ANum of size 12 bytes. Suppose an index file with records consisting of two fields, ANum value and the record pointer the corresponding student record, is built and stored on the same disk. Assume that the records of data file and index file are not split across disk blocks. The number of blocks in the index file is _____

Solution:

Given: Total Number of Records = 1,50,000

Block Size = 4096 Bytes

Record pointer = 7 Bytes

Candidate Key, ANum = 12 Bytes

Formula:

One Record Size = Record pointer + Key Size

Record size = 7 Bytes + 12 Bytes = 19 Bytes

Number of Records into the block = Block Size/ Record Size

Number of Records into the block = 4096 Bytes/ 19 Bytes = 215

Number of Blocks need to store all the records = Total Number of Records/ Number of Records into the block

Number of Blocks need to store all the records = 150000/ 215 = 697.67 = 698

***Transaction Processing Concept:** Transaction System, Testing of Serializability, Serializability of Schedules, Conflict and View Serializable Schedule, Recoverability, Recovery from Transaction Failure, Log Based Recovery, Deadlock Handling.

Transaction: A transaction is a single logical unit of work which accesses and possibly modifies the contents of a database. (Transaction is generally represent change in database.)

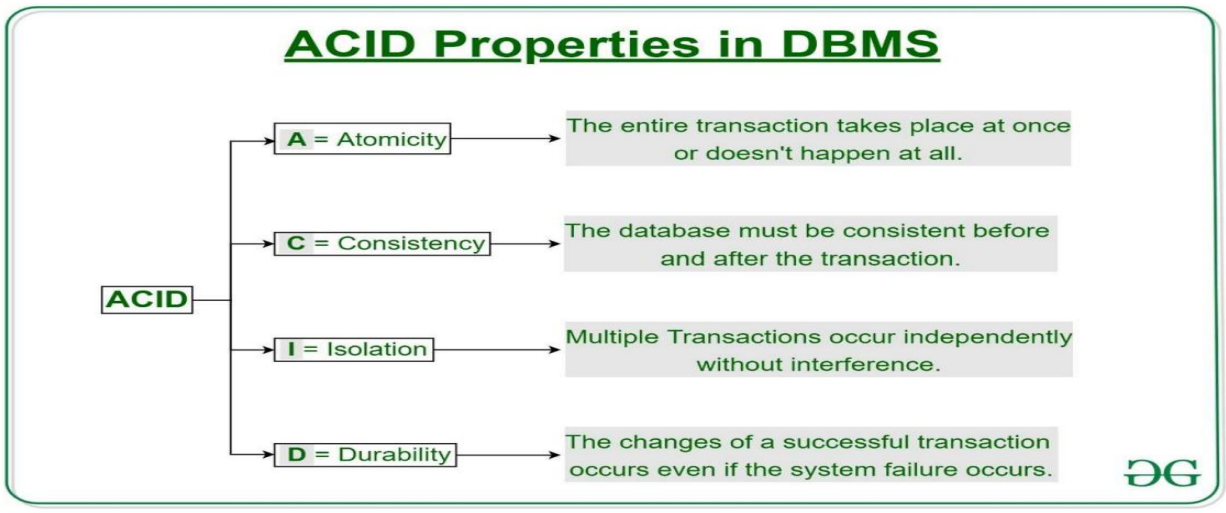
Transactions access data using read and write operations. In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called ***ACID** properties.

A: 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.

C: 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.

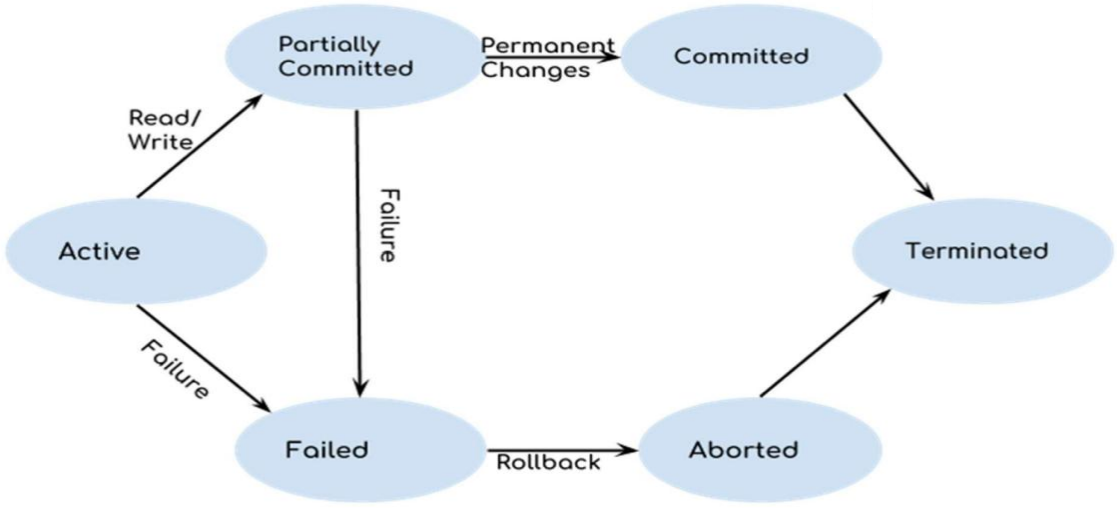
I: Isolation: This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed.

D: 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.



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

*****Transaction States:**



Active State: As we have discussed in the DBMS transaction introduction that a transaction is a sequence of operations. If a transaction is in execution then it is said to be in active state.

Failed State: If a transaction is executing and a failure occurs, either a hardware failure or a software failure then the transaction goes into failed state from the active state. (Power failure)

Partially Committed State: As we can see in the above diagram that a transaction goes into "partially committed" state from the active state when there are read and write operations present in the transaction. A transaction contains number of read and write operations. Once the whole transaction is successfully executed, the transaction goes into partially committed state where we have all the read and write operations performed on the main memory (local memory) instead of the actual database. The reason why we have this state is because a transaction can fail during execution so if we are making the changes in the actual database instead of local memory, database may be left in an inconsistent state in case of any failure. This state helps us to rollback the changes made to the database in case of a failure during execution.

Committed State: If a transaction completes the execution successfully then all the changes made in the local memory during partially committed state are permanently stored in the database. You can also see in the above diagram that a transaction goes from partially committed state to committed state when everything is successful.

Aborted State: if a transaction fails during execution then the transaction goes into a failed state. The changes made into the local memory (or buffer) are rolled back to the previous consistent state and the transaction goes into aborted state from the failed state. Refer the diagram to see the interaction between failed and aborted state.

Terminated State: This is the last state in the life cycle of a transaction. After entering the committed state or aborted state, the transaction finally enters into a terminated state where its life cycle finally comes to an end.

Schedule: It is a chronological execution sequence of multiple transaction.

Serial Schedules: In Serial schedule, a transaction is executed completely before starting the execution of another transaction. In other words, you can say that in serial schedule, a transaction does not start execution until the currently running transaction finished execution. This type of execution of transaction is also known as serial execution.

Example: Consider the following schedule involving two transactions T1 and T2.

T_1	T_2
R(A)	
W(A)	
R(B)	
	W(B)
	R(A)
	R(B)

This is a serial schedule since the transactions perform serially in the order T1 —> T2

There are two transactions T1 and T2 executing serially one after the other.

Transaction T1 executes first.

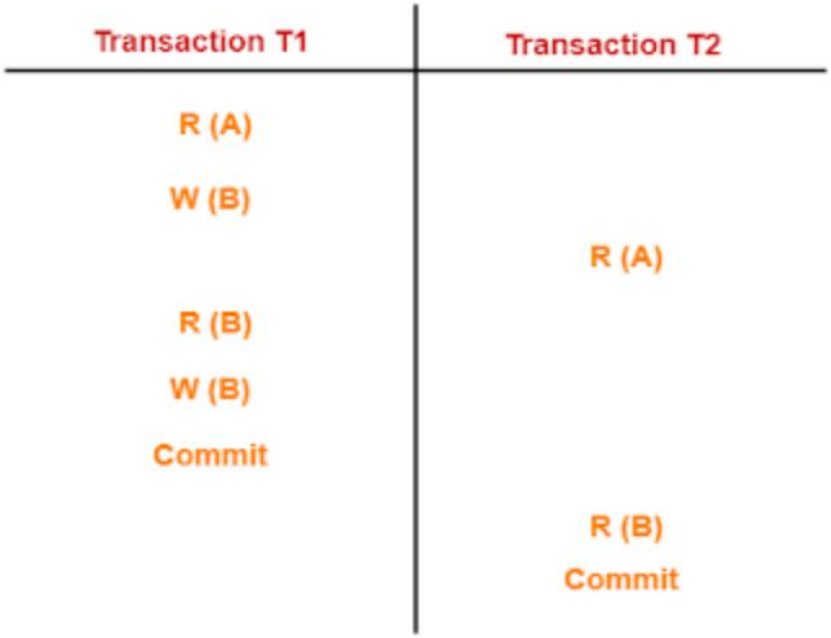
After T1 completes its execution, transaction T2 executes.

So, this schedule is an example of a Serial Schedule.

Advantage: Security and Consistent Results

Disadvantage: Waiting Time (When T1 is in execution T2 waits)

Parallel Schedule: Unlike the serial schedule where one transaction must wait for another to complete all its operation, in the non-serial/parallel schedule, the other transaction proceeds without waiting for the previous transaction to complete. This sort of schedule does not provide any benefit of the concurrent transaction.



In non-serial/parallel schedules,

Multiple transactions execute concurrently.

Operations of all the transactions are inter leaved or mixed with each other.

Advantage: No Waiting Time, Performance

Disadvantage: Parallel schedules are NOT always-

Consistent

Recoverable

Cascade less

Strict

*****READ/ WRITE OPERATIONS**

FOR SAME VARIABLE (A)		
R (A)	**R (A)**	**Non - Conflict Pair**
R (A)	**W (A)**	**Conflict Pair**
W (A)	**R (A)**	**Conflict Pair**
W (A)	**W (A)**	**Conflict Pair**

FOR DIFFRENT VARIABLES (A,B): No conflict occurs between two different variables		
R (B)	**R (A)**	**Non - Conflict Pair**
W (B)	**R (A)**	**Non - Conflict Pair**
R (B)	**W (A)**	**Non - Conflict Pair**
W (A)	**W (B)**	**Non - Conflict Pair**

Conflict Equivalent Schedules: Swap Adjacent Non Conflict Pair only.

Date / / Page no:

Conflict Equivalent Schedules

Ques 1 :- $S = S'$ To Prove.

S

T_1	T_2
R(A)	
W(A)	
	R(A)
R(B)	W(A)

$\Rightarrow$

T_1	T_2
R(A)	
W(A)	
R(B)	R(A)
	W(A)

Adjacent non-conflict Pair "Swap"

$\Downarrow\Downarrow$

S'

T_1	T_2
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)

$\Leftarrow$

T_1	T_2
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)

Hence, Proved
$S = S'$ { Conflict Equivalent schedule }.

Date: / / Page no:

Ques 2 :- To prove $S = S'$

S

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	

=

S'

T_1	T_2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)

Solution :-

S

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B	

Swap non-conflict adjacent Pair

T_1	T_2
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
W(B)	

T_1	T_2
R(A)	
W(A)	
R(B)	
	R(A)
W(B)	
	W(A)

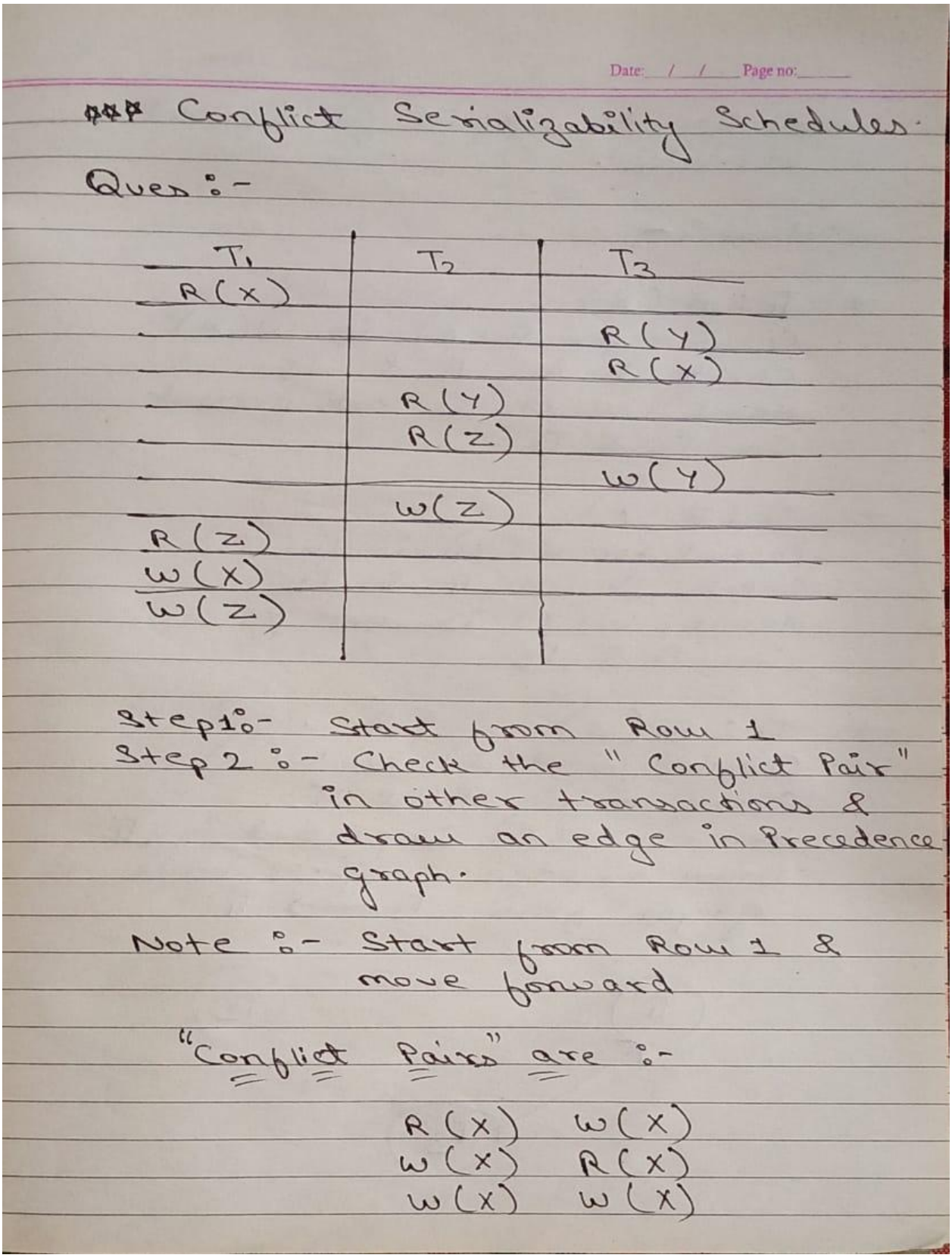
Here, we have two non-conflict adjacent pairs Swap it

S'

T_1	T_2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)

Hence $S = S'$ {conflict equivalent Schedules}

***Conflict Serializability Schedules:** Check for the Conflict Pairs only.



*** Conflict Serializability Schedules.

Ques :-

T_1	T_2	T_3
R(X)		
		R(Y)
		R(X)
	R(Y)	
	R(Z)	
		W(Y)
	W(Z)	
R(Z)		
W(X)		
W(Z)		

Step 1 :- Start from Row 1

Step 2 :- Check the "Conflict Pair" in other transactions & draw an edge in Precedence graph.

Note :- Start from Row 1 & move forward

"Conflict Pairs" are :-

R(X)	W(X)
W(X)	R(X)
W(X)	W(X)

Date: / / Page no:

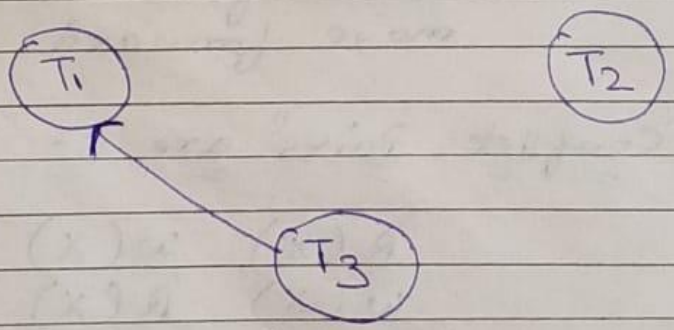
Note :- No conflict for different variables.

Solution :-

→ $T_1 : R(X)$ {Row 1}
Conflict for $R(X)$ is $W(X)$
Check it in T_2 & T_3
Hence, $W(X)$ is not present in T_2 & T_3.

→ $T_3 : R(Y)$ {Row 2}
Conflict for $R(Y)$ is $W(Y)$
Check it in T_1 & T_2
Hence, $W(Y)$ is not present in T_1 & T_2

→ $T_3 : R(X)$ {Row 3}
Conflict for $R(X)$ is $W(X)$
Check it in T_1 & T_2
Hence, $W(X)$ is present in T_1
Draw an edge in precedence graph from $T_3 \longrightarrow T_1$



Date: / / Page no:

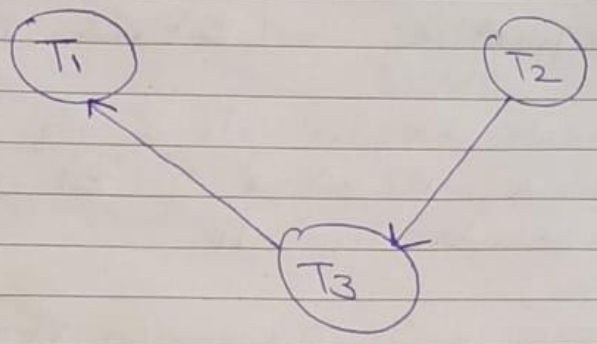
→ $T_2 : R(Y)$ {Row 4}

Conflict for $R(Y)$ is $W(Y)$

Check it in T_1 & T_3

Hence, $W(Y)$ is present in T_3

Draw an edge in Precedence Graph from $T_2 \longrightarrow T_3$



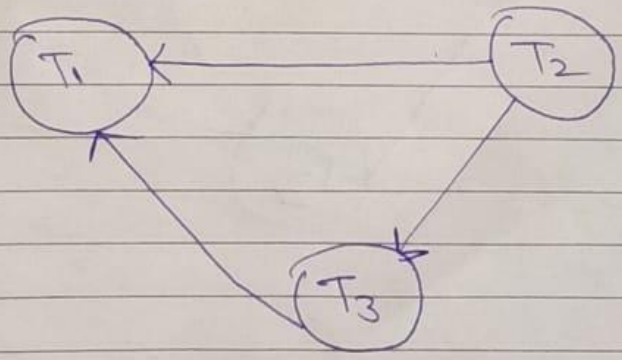
→ $T_2 : R(Z)$ {Row 5}

Conflict for $R(Z)$ is $W(Z)$

Check it in T_1 & T_3

Hence, $W(Z)$ is present in T_1

Draw an edge in Precedence Graph from $T_2 \longrightarrow T_1$



Date: / / Page no:

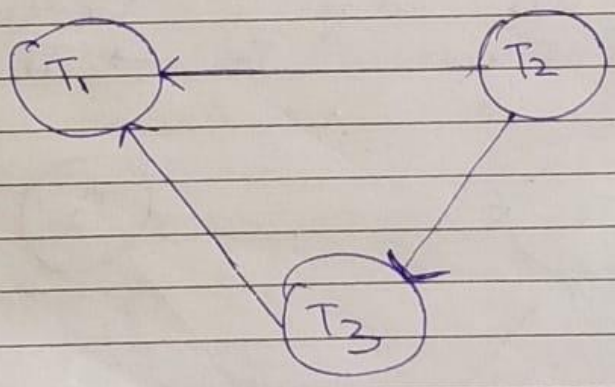
→ T_3 : w(y) {Row 6}
Conflict of w(y) is R(y) & w(y)

Check it in T_1 & T_2
Hence, R(y) & w(y) is not present in T_1 & T_2

→ T_2 : w(z) {Row 7}
Conflict of w(z) is R(z) & w(z)

Check it in T_1 & T_3
Hence, R(z) & w(z) both are present in T_1

Draw an edge in Precedence Graph from $T_2 \longrightarrow T_1$ which is already drawn. ∴ no need to draw it again.



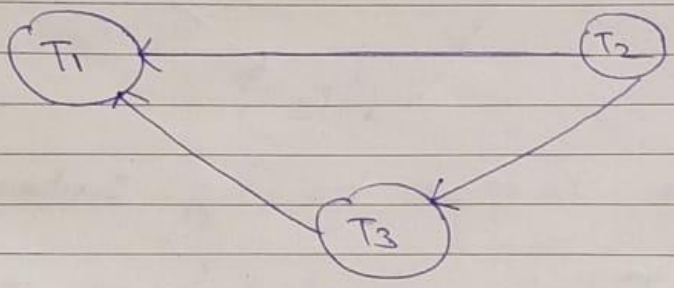
Date: / / Page no:

→ $T_1 : R(z)$ {Row 8}
Nothing left in T_2 & T_3
So, can't compare.

→ $T_1 : w(x)$ {Row 9}
can't compare

→ $T_1 : w(z)$ {Row 10}
can't compare.

Final Precedence Graph is



Now, check for loop :-

If
Starting & ending point is same there is loop

Else
No loop.

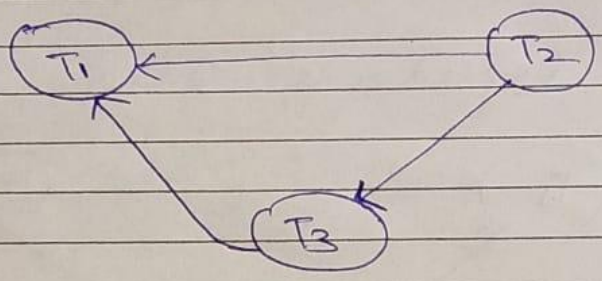
In the above precedence graph, there is no loop exists.

Date: / / Page no:

Note :-

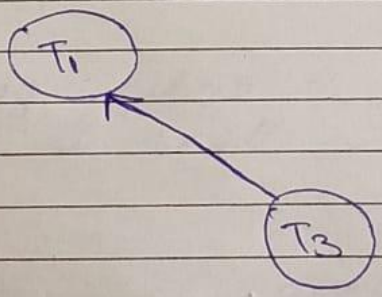
If no loop → Conflict Serializable Schedule
↓
Serializable
↓
Consistent.

Prove :- How it is Serializable?



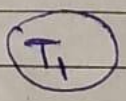
Search Vertex having Indegree '0'

T_2 having indegree Zero
∴ Erase T_2 first.



Search vertex having Indegree "0"

T_3 having indegree zero
∴ Erase T_3



Date: / / Page no:

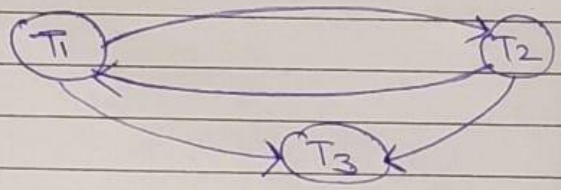
Serializability is :-

$T_2 \longrightarrow T_3 \longrightarrow T_1$

Ques 2 :-

T_1	T_2	T_3
R(B)		
		R(C)
R(A)		
	W(A)	
W(A)		
	W(B)	
		W(A)
W(B)		
		W(B)
		W(C)

Solution :- Find Precedence Graph



Loop is here.

Note :-

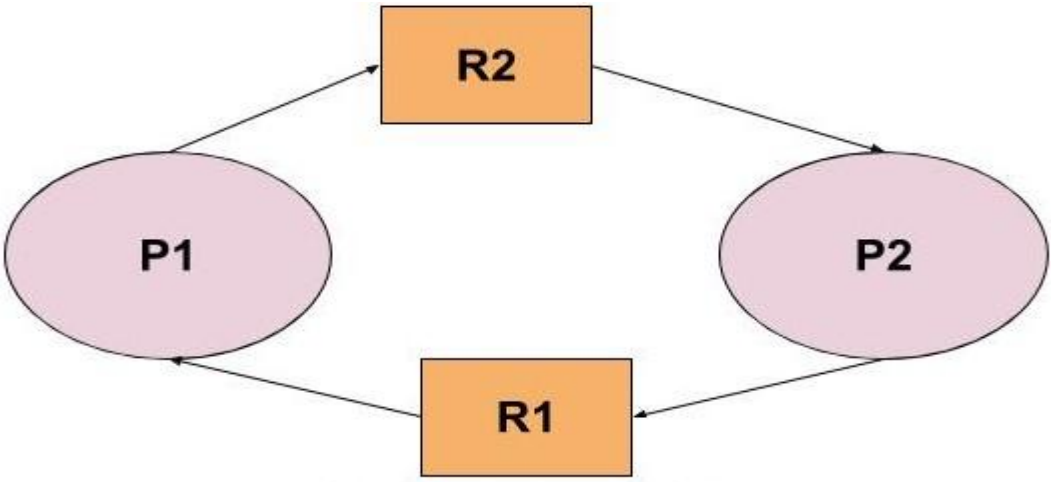
If Loop :- not conflict Serializable Schedule

No vertex is there having indegree zero :: not serializable.

***DEADLOCK

Deadlock is a critical challenge in database management systems (DBMS) that can cause system failure and impact overall performance. The potential for deadlocks increases in a multi-user environment, where multiple transactions compete for shared resources.

A deadlock in DBMS (Database Management System) occurs when two or more transactions are stuck in a never-ending or indefinite waiting state. Each transaction needs a resource that another transaction holds, which is referred to as the locking of resources, thus, creating a deadlock situation.



Process P1 is holding R1 and waiting for R2 while Process P2 is holding R2 and waiting for R1.

The figure shows an example where process P1 holds the resource R1 and requesting for R2. Similarly, P2 holds the resource R2 and requesting for R1 . Both processes are waiting for each other to release the resource to proceed further.

Necessary Conditions for Deadlock

Mutual Exclusion: It arises when only one process gets allocated to a single resource at a time. If multiple processes request the same resources simultaneously, deadlock can occur if they are unable to process unless they can lock that resource. In the example that we discussed, cars A and B got stuck in a condition where only one of them could go through the lane, creating a mutual exclusion condition.

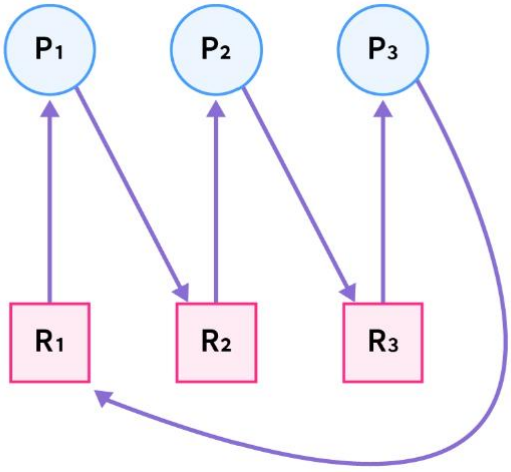
Hold and Wait: Processes hold resources while waiting for additional resources, creating a situation where everyone is holding onto something and waiting for more. For example, process P1 holds resource R1 and needs resource R2 (held by process P2), and P2 needs R1 (held by P1).

No Preemption: Resources cannot be forcibly taken away from processes, meaning once you hold some resource, you keep it until you're done utilizing it.

If process P1 is using a resource R1 and a high priority process P2 enters and requests for the resource R1, than process P1 will not stopped and the resource R1 not allocated to P2.

Circular wait: Processes form a circular chain of resource dependencies, where each process waits for a resource held by another process, resulting in a never-ending cycle. We discussed this condition in the example, where vehicles A and B were "circularly" waiting for each other to release the lane for them to proceed.

In circular wait, two or more processes wait for resources in a circular order.



Deadlock Handling in DBMS

Example:

There are **four** ways to deal with deadlock/ deadlock handling:

1. **Deadlock Prevention in DBMS:** To prevent deadlocks before they can happen. For instance, only one car can cross an intersection at a time.
2. **Deadlock Avoidance in DBMS:** Similar to having a traffic controller managing the flow of cars, in DBMS, a system monitors potential deadlocks and decides which processes can proceed to prevent deadlock situations.
3. **Deadlock Ignorance in DBMS:** Imagine cars driving without specific rules or precautions. In DBMS, deadlock ignorance means not taking specific measures. It's like allowing cars to navigate without traffic lights or controllers and dealing with any issues as they arise.
4. **Deadlock Detection in DBMS:** If there's a traffic accident, it's detected, and authorities respond. Similarly, in DBMS, deadlock detection identifies when a deadlock has occurred, and the system takes action to resolve it.

***** LOG BASED RECOVERY**

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

LOG Records are:

<Ti Start> When transaction Ti Starts

<Ti, Xj, V1, V2> Transaction Ti has performed, Xj is variable, V1 is old value, V2 is new value

<Ti Commit> To save Ti permanently

<Ti Abort> To abort Ti

IT IS OF TWO TYPES:-

IMMEDIATE DATABASE MODIFICATION	DEFFERED DATABASE MODIFICATION
In an immediate update, the changes are applied directly/immediately to the database.	In a deferred update, the changes are not applied immediately to the database.
The log file contains both old as well as new values.	The log file contains only new value.
Represented as: **<Ti, Xj, V1, V2>**	Represented as: **<Ti, Xj, V2>**

EXAMPLE: Immediate		**EXAMPLE: Deffered**	
Given: A = 5, B=7		**Given: A = 600, B = 900, C = 1200**	
T1	<T1 Start>	**T0**	**T1**
R (A)	No log record for read operation	**R (A)**	**R (C)**
R (B)	No log record for read operation	**A = A - 50;**	**C = C - 100**
A = A + B;	<T1, A, 5, 12>	**W (A)**	**W (C)**
W (A)	<T1 Commit>	**R (B)**	
		B = B + 50;	
		W (B)	
		<T0 Start>	
		No log record for read operation	
		<T0, A, 550>	
		<T0 Commit>	
		No log record for read operation	
		<T0, B, 950>	
		<T0 Commit>	
		<T1 Start>	
		No log record for read operation	
		<T1, C, 1100>	
		<T1 Commit>	

REDO / UNDO OPERATIONS

If failure occurs after <Ti, Xj, V1, V2> or <Ti, Xj, V2> than perform

REDO: To Restart

If failure occurs after commit than perform

UNDO: To Overwrite

***VIEW IN DATABASE

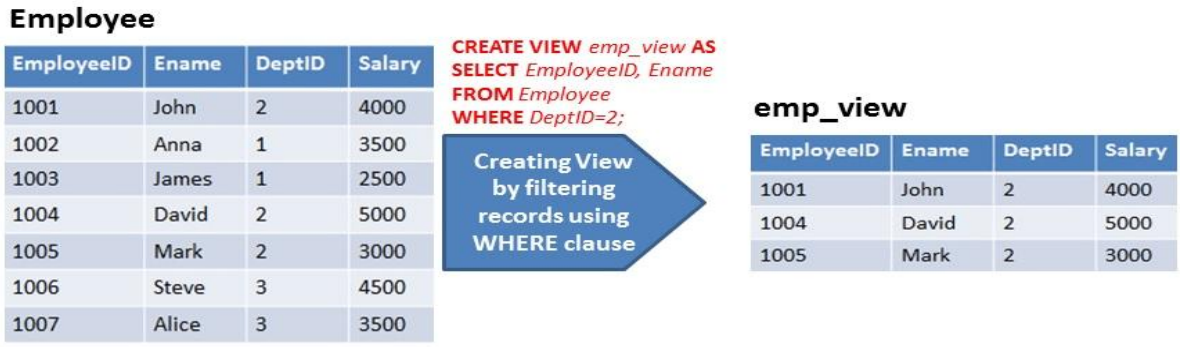
VIEW TABLE: View table is a logical table based on one or more original table/ base table.

Characteristics:

- Data is in the virtual table not stored permanently
- It doesn't contain any data of its own.
- It doesn't take any memory.
- Views are used for security purpose because they provide encapsulation.
- View table only displays selected data
- View is also known as "Stored Select Statement"

Types of Views:

1. Simple View: View table formed with the help of one base table. Does not stores data.



2. Complex View: View table formed with the help of two or more base tables. Does not stores data.

3. Materialized View: Like a physical table that stores query output/ intermediate results. It stores Data. This is basically used in Data ware House.

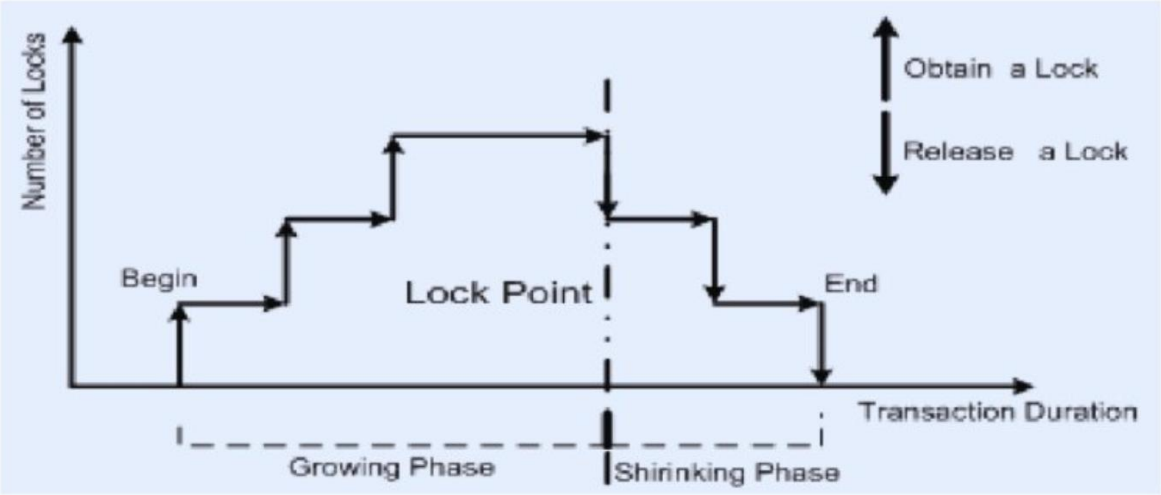
UPDATABLE VIEW	NON UPDATABLE VIEW
Here, you can use: ● INSERT ● UPDATE ● DELETE	If a view is non updatable such as INSERT, UPDATE, DELETE are illegal and are rejected.

For a view to be updatable, there must be a one-to-one relationship between the rows in the view and the rows in the underlying table. To be more specific, a view is updatable if it not contains any of the following: ● Aggregate functions (SUM(), MIN(), MAX(), COUNT(), AVG()) ● DISTINCT ● GROUP BY ● HAVING ● UNION or UNION ALL ● Sub-query in the select and where clause ● Left Join, Outer Join	Any attempt to modify data through a non updatable view will results in error To be more specific, a view is non updatable if it contains any of the following: ● Aggregate functions (SUM(), MIN(), MAX(), COUNT(), AVG()) ● DISTINCT ● GROUP BY ● HAVING ● UNION or UNION ALL ● Subquery in the select and where clause ● Left Join, Outer Join

*****TWO PHASE LOCKING (2PL)**

Two-Phase Locking (2PL) is a concurrency control mechanism used in database management systems to ensure the consistency of transactions. The primary goal of 2PL is to prevent conflicts between concurrent transactions that could lead to data inconsistencies. The protocol is divided into two phases: the growing phase and the shrinking phase.

Two-phase locking (2PL) is a protocol used in database management systems to control concurrency and ensure transactions are executed in a way that preserves the consistency of a database. It's called "two-phase" because, during each transaction, there are two distinct phases: the Growing phase and the Shrinking phase.



Phases:

Growing Phase: During this phase, a transaction can obtain (acquire) any number of locks as required but cannot release any. This phase continues until the transaction acquires all the locks it needs and no longer requests.

Locks can be of two types: **shared locks and exclusive locks**.

- **Shared locks** are used when a transaction wants to **read** a resource.
- **Exclusive locks** are used when a transaction wants to **write** to a resource.

Shrinking Phase: Once the transaction releases its first lock, the Shrinking phase starts. During this phase, the transaction can release but not acquire any more locks.

Lock Point: The exact moment when the transaction switches from the Growing phase to the Shrinking phase (i.e. when it releases its first lock) is termed the lock point.

The primary purpose of the Two-Phase Locking protocol is to ensure conflict serializability, as the protocol ensures a transaction does not interfere with others in ways that produce inconsistent results.

Distributed Database: Introduction To Distributed Database, Data Fragmentation And Data Replication

Distributed Database: A distributed database is a collection of multiple, logically interrelated databases distributed over a computer network and managed as a single database.

A distributed database refers to a collection of databases scattered across multiple sites over a network. Think of this system as a Multinational Corporation, whose offices and systems are distributed across the globe, yet they work harmoniously for a common purpose.

In a Distributed Database Management System (Distributed DBMS or DDBMS), data, processes, and interface components of a system are divided across multiple locations to allow independent functioning. Different databases are stored across multiple computers, and accordingly, the workload processing is distributed. This network architecture delivers much better performance and increased reliability compared to Centralized Database Management Systems.

Features of Distributed Database Management System

A Distributed Database Management System (DDBMS) carries the full functionality of a standard DBMS and provides transparent management of distributed and replicated data. It also:

- Improves reliability & data availability through distributed transactions.
- Provides an easier and more economical system for database expansion.
- Maintains confidentiality and data integrity of databases.
- Is hardware independent.
- Delivers quick responses to user queries through efficient distribution of data.
- Gives users a simple interface to open, read/write records, and close files.
- Comes with declarative query capabilities, transaction management, and integrity enforcement.

Fragmentation and Replication In Distributed Database

Data Fragmentation: Data fragmentation involves breaking down a database into smaller, more manageable pieces or fragments. There are several types of data fragmentation:

Horizontal Fragmentation:

- Divides the rows of a table into subsets or fragments.
- Each fragment contains a subset of rows.

Vertical Fragmentation:

- Divides the columns of a table into subsets or fragments.
- Each fragment contains a subset of columns.

Hybrid Fragmentation:

- Combines both horizontal and vertical fragmentation.

Data Replication: Data replication involves creating and maintaining copies of the same data on multiple nodes in a distributed database system. Replication provides benefits such as increased availability and fault tolerance. There are different types of data replication:

Full Replication:

- All nodes in the distributed system have a complete copy of the entire database.
- Ensures high availability but requires more storage.

Partial Replication:

- Only a subset of the data is replicated across nodes.
- Balances availability and storage requirements.

Selective Replication:

- Only specific portions of the database are replicated based on certain criteria.
- Allows for fine-grained control over replication.

Homogeneous Distributed Database:	**Heterogeneous Distributed Database:**
A homogeneous distributed database system is one in which all nodes (database servers or sites) have the same database management system (DBMS) software and similar hardware configurations.	A heterogeneous distributed database system, on the other hand, is one in which, nodes may have different database management system (DBMS) software, hardware platforms, or data models.

Advantages:

- Scalability: Distributes data to handle increased load and volume.
- High Availability: Redundant copies ensure system remains accessible despite node failures.
- Fault Tolerance: System resilience to node failures enhances overall reliability.
- Improved Performance: Parallel processing and data localization enhance query response times.

- Geographic Distribution: Stores data closer to users, reducing latency in dispersed systems.
- Flexibility: Supports diverse data models and technologies across nodes.
- Data Security: Redundancy and replication enhance data security and resilience.
- Cost Efficiency: Utilizes cost-effective hardware by distributing the load.

Disadvantages:

Complexity: Management and coordination across distributed nodes can be intricate.

Costs: Initial setup and maintenance costs can be higher due to complexity.

QUESTIONS FOR PRACTICE

Refer Assignment 2

Set 1: Ques 1, 2, 3, 4, 5

Set 2: Ques 4, 5

Set 3: Ques 1, 2, 3

Set 4: Ques 1, 2, 4

List of Important Topics

Lossless, Lossy, Dependency Preserving

Indexing with types (B, B+ TREE)

Transaction States

ACID properties

View (Updatable & Non Updatable)

Conflict Schedules, precedence graph, recoverability

Log based recovery

Deadlock handling

2PL: Growing and Shrinking Phase

Distributed Database