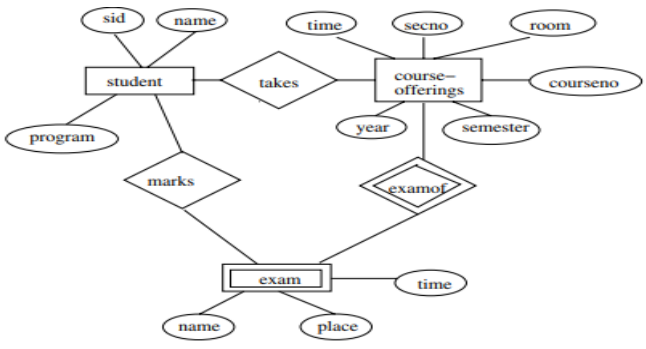
ABES Engineering College, Ghaziabad
DBMS Makeup Test Solution (KCS-501)
1(a)

ER Diagram stands for Entity Relationship Diagram, also known as ERD is a diagram that displays the relationship of entity sets stored in a database. In other words, ER diagrams help to explain the logical structure of databases.

The components of ER diagram are as follows −

- **Entity**: It may be an object, person, place or event that stores data in a database. In a relationship diagram an entity is represented in rectangle form. For example, students, employees, managers, etc.
- **Attributes**: It is the name, thing etc. These are the data characteristics of entities or data elements and data fields.
- **Relationship**: It is used to describe the relation between two or more entities. It is represented by a diamond shape For Example, students study in college and employees work in a department.
- **Weak entity**: The entity sets which do not have sufficient attributes to form a primary key are known as **weak entity.**
- **Strong entity**: A strong entity is an entity that is not dependent on any other entity. It has a primary key, or a table includes a primary key.
- **Simple attribute**: An attribute that cannot be further subdivided into components is a simple attribute. Example: The roll number of a student, the id number of an employee.
- **Key attribute**: A key in DBMS is an attribute or a set of attributes that help to uniquely identify a tuple (or row) in a relation (or table)
- **Composite attribute**: An attribute that can be split into components is a composite attribute. Example: The address can be further split into house number, street number, city, state, country, and pin code, the name can also be split into first name middle name, and last name.
- **Derived attribute**: A derived attribute is an attribute or property in a table that has been calculated or derived using other attributes in the database.
- **Multivalued attribute**: multivalued attribute of an entity is an attribute that can have more than one value associated with the key of the entity. For example, a large company could have many divisions, some of them possibly in different cities.



1(b)

Super key: It is an attribute set that can uniquely identify a tuple. A super key is a superset of a candidate key. For example: In EMPLOYEE table, for(EMPLOEE_ID, EMPLOYEE_NAME), the name of two employees can be the same, but their EMPLYEE_ID can't be the same. Hence, this combination can also be a key. The super key would be EMPLOYEE-ID (EMPLOYEE_ID, EMPLOYEE-NAME).

Primary key: It is the first key used to identify one and only one instance of an entity uniquely. An entity can contain multiple keys, as we saw in the PERSON table. The key which is most suitable from those lists becomes a primary key. In the EMPLOYEE table, ID can be the primary key since it is unique for each employee. In the EMPLOYEE table, we can even select License_Number and Passport_Number as primary keys since they are also unique.

Candidate key: A candidate key is an attribute or set of attributes that can uniquely identify a tuple. Except for the primary key, the remaining attributes are considered a candidate key. The candidate keys are as strong as the primary key. For example: In the EMPLOYEE table, id is best suited for the primary key. The rest of the attributes, like SSN, Passport_Number, License_Number, etc., are considered a candidate key.

The overall design of the database is called the database schema.

2 (a)

(i) Only 1 tuple is returned.

$$\prod_X \left(\sigma_{(P.Y=R.Y \wedge R.V=V2)}(P \times R)\right)$$

Returns

X
X2

$$\prod_X \left(\sigma_{(Q.Y=R.Y \wedge Q.T>2)}(Q \times R)\right)$$

Returns

X
X1

$$\prod_X \left(\sigma_{(P.Y=R.Y \wedge R.V=V2)}(P \times R)\right) - \prod_X \left(\sigma_{(Q.Y=R.Y \wedge Q.T>2)}(Q \times R)\right)$$

Output:

X
X2

(ii) Query:

$$\Pi_{empId}\,(employee) - \Pi_{empId}\,(employee \bowtie_{(empId=eID)\wedge(empAge\leq depAge)} dependent)$$

(iii) Query in TRC:

$$\{e.name \mid employee(e) \wedge$$
$$(\forall x)[\neg employee(x) \vee$$
$$x.\mathrm{sup}\,ervisorName \neq e.name \vee$$
$$x.sex =' male']\}$$

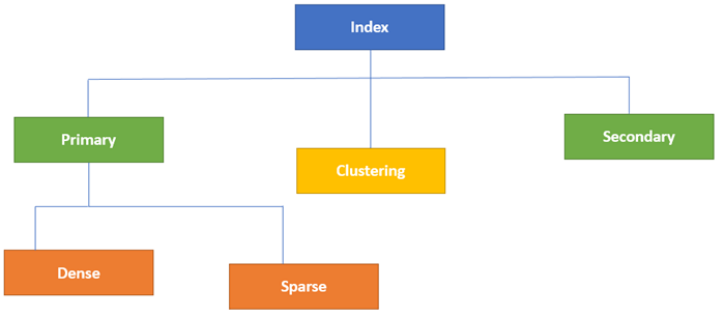
2(b)

Purpose of Indexing:

- Indexing is used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.

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

Types of Indexing:



Indexing in Database is defined based on its indexing attributes. Two main types of indexing methods are:

- Primary Indexing
- Secondary Indexing
- Clustered Indexing

Primary Index in DBMS

Primary Index is an ordered file which is fixed length size with two fields. The first field is the same a primary key and second, filed is pointed to that specific data block. In the primary Index, there is always one to one relationship between the entries in the index table.

The primary Indexing in DBMS is also further divided into two types.

- Dense Index
- Sparse Index

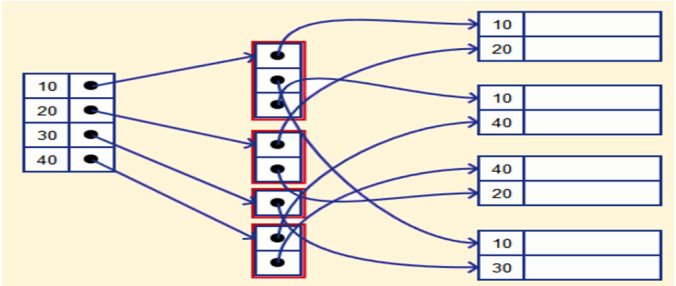
Secondary Index in DBMS

The secondary Index in DBMS can be generated by a field which has a unique value for each record, and it should be a candidate key. It is also known as a non-clustering index.

This two-level database indexing technique is used to reduce the mapping size of the first level. For the first level, a large range of numbers is selected because of this; the mapping size always remains small.

Example: In a bank account database, data is stored sequentially by acc_no; you may want to find all accounts in of a specific branch of ABC bank.

Here, you can have a secondary index in DBMS for every search-key. Index record is a record point to a bucket that contains pointers to all the records with their specific search-key value.



Clustering Index in DBMS

In a clustered index, records themselves are stored in the Index and not pointers. Sometimes the Index is created on non-primary key columns which might not be unique for each record. In such a situation, you can group two or more columns to get the unique values and create an index which is called clustered Index. This also helps you to identify the record faster.

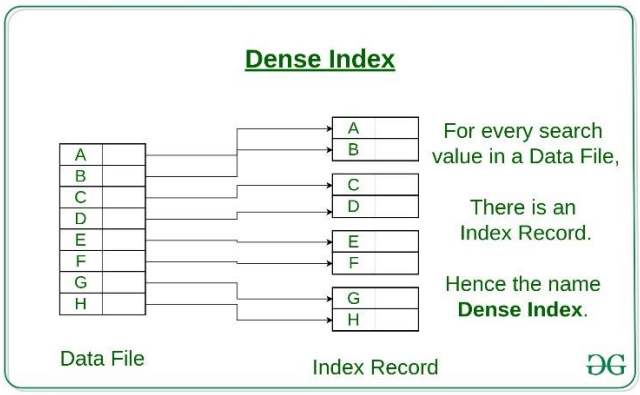
Example:

Let's assume that a company recruited many employees in various departments. In this case, clustering indexing in DBMS should be created for all employees who belong to the same dept.

It is considered in a single cluster, and index points point to the cluster as a whole. Here, Department _no is a non-unique key.

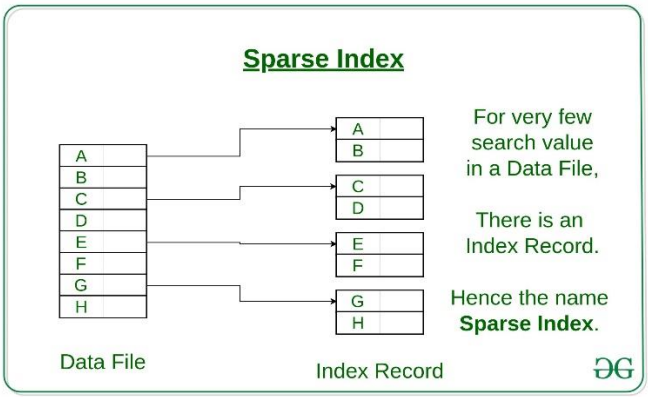
Dense Index:

- For every search key value in the data file, there is an index record.
- This record contains the search key and also a reference to the first data record with that search key value.



Sparse Index:

- The index record appears only for a few items in the data file. Each item points to a block as shown.
- To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
- We start at that record pointed to by the index record, and proceed along with the pointers in the file (that is, sequentially) until we find the desired record.
- Number of Accesses required=$\log_2(n)+1$, (here n=number of blocks acquired by index file)



3(a)

Solution : 3(a) : $R(A, B, C, D, E, F, G, H)$

FD : $\{CH \to G, A \to BC, B \to CFH, E \to A, F \to EG\}$

Step ① : Identify all possible candidate keys

Essential Attributes : $\{D\}$

$\{D\}^+ = \{D\}$

find $\{AD\}^+ = \{A, B, C, D, F, H, E, G\}$

AD is candidate key.

∵ A is determined by E $(E \to A)$

∴ ED is another candidate key.

∵ E is determined by F $(F \to E)$

∴ FD is another candidate key.

∵ F is determined by B $(B \to F)$

∴ BD is another candidate key.

Candidate Keys : $\{AD, ED, FD, BD\}$

Prime Attributes : $\{A, B, D, E, F\}$

Non prime Attributes : $\{C, G, H\}$

Step ② : Identify highest normal form.

Check for BCNF : If $A \to B$ holds, then A must be super key (SK).

Here $CH \to G$ holds but CH is not SK

∴ R is not in BCNF.

Check for 3NF : If $A \to B$ holds, then either A is SK or B is prime attribute

Here $CH \to G$ holds but neither CH is SK nor G is prime attribute

∴ R is not in 3NF.

check for 2NF : If $A \to B$ holds, A is proper subset of CK & B is non prime attribute (condition of partial dependency).

Here $A \to C$ holds, A is proper subset of CK (AD) & C is non prime attribute (partial dependency exist)

∴ R is not in 2NF.

⟹ Highest normal form : 1NF.

3(b)

(i) You are given the table below for a relation R(A,B,C,D,E). You do not know the functional dependencies for this relation.

A	B	C	D	E
'a'	122	1	's1'	'a'
'e'	236	4	'e2'	'b'
'a'	199	1	'b5'	'c'
'b'	213	2	'z8'	'd'

Suppose this relation is decomposed into the following two tables: R1(A,B,C,D) and R2(A,C,E).

R1

A	B	C	D
'a'	122	1	's1'
'e'	236	4	'e2'
'a'	199	1	'b5'
'b'	213	2	'z8'

R2

A	C	E
'a'	1	'a'
'e'	4	'b'
'a'	1	'c'
'b'	2	'd'

To find whether decomposed relations are lossless join decomposition or not, we need to join these relations:

$R1 \bowtie R2$

A	B	C	D	E
'a'	122	1	's1'	'a'
'e'	236	4	'e2'	'b'
'a'	199	1	'b5'	'c'
'b'	213	2	'z8'	'd'
'a'	122	1	's1'	'a'
'a'	199	1	'b5'	'c'

From the result, it is clear that we have got more number of rows as compared to original relation. Therefore, this decomposition is not lossless.

(ii)

Solution: 3(b)(ii) $R(A, B, C, D, E, f, G)$

FD: $\{AD \to Bf, CD \to EGC, BD \to f, E \to D, f \to C, D \to f\}$

To find minimal / canonical cover.

Step (1): Decompose all functional dependency.

FD: $\{AD \to B, AD \to f, CD \to E, CD \to G, CD \to C, BD \to f, E \to D, f \to C, D \to f\}$

Step (2): Remove Extraneous FD's.

for $AD \to B$, $\{AD\}^+ = \{A, D, B, f, C, E, G\}$

$\{AD\}^f = \{A, D, f, C, G, E\}$

Both are different, so $AD \to B$ is essential FD & hence cannot be removed.

In similar fashion, check for all FD's.

∴ $CD \to E$, $AD \to f$, $BD \to f$ are extraneous FD.

FD: $\{AD \to B, CD \to E, CD \to G, E \to D, f \to C, D \to f\}$

Step (3): Remove Extraneous attributes in ~~ev~~ FD.

Remove D from $CD \to E$ & $CD \to G$

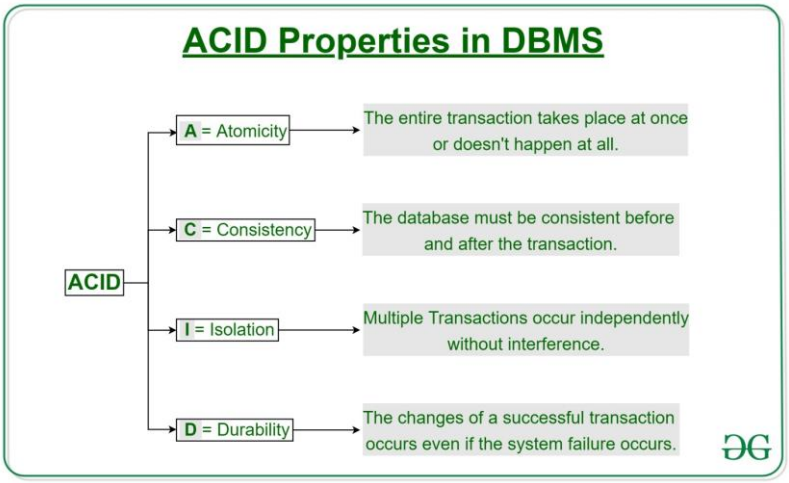
finally recombine, so minimal cover is

FD: $\{AD \to B, D \to EGf, f \to C, E \to D\}$.

4(a)

A transaction 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.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X − 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

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".

T	T''
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y – 50 Write (Y)	Read (X) Read (Y) Z: = X + Y Write (Z)

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).

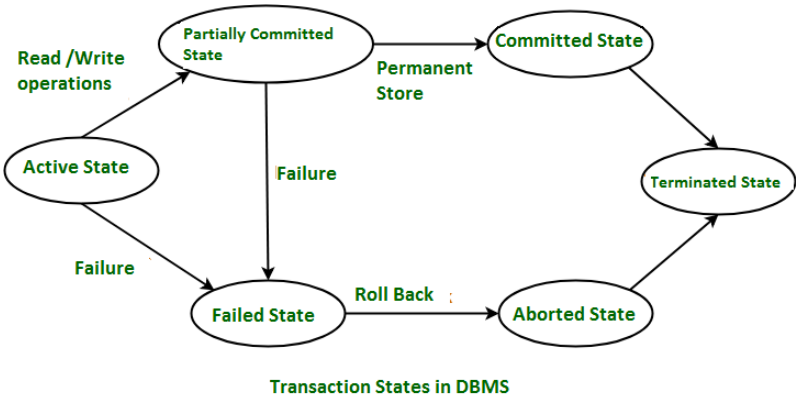
This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

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.

Transaction States in DBMS

States through which a transaction goes during its lifetime. These are the states which tell about the current state of the Transaction and also tell how we will further do the processing in the transactions. These states govern the rules which decide the fate of the transaction whether it will commit or abort.



Transaction States in DBMS

These are different types of Transaction States:

Active state

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

- The active state is the first state of every transaction. In this state, the transaction is being executed.
- **For example:** Insertion or deletion or updating a record is done here. But all the records are still not saved to the database.

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".

- In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database.
- In the total mark **calculation example, a** final display of the total marks step is executed in this state.

Committed

A transaction is said to be in a committed state if it executes all its operations successfully. In this state, all the effects are now permanently saved on the database system.

Failed state

- If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
- In the **example** of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

Aborted

- If any of the checks fail and the transaction has reached a failed state then the database recovery system will make sure that the database is in its previous consistent state. If

not then it will abort or roll back the transaction to bring the database into a consistent state.

- If the transaction fails in the middle of the transaction then before executing the transaction, all the executed transactions are rolled back to its consistent state.
- After aborting the transaction, the database recovery module will select one of the two operations:
    1. Re-start the transaction
    2. Kill the transaction

Terminated State –

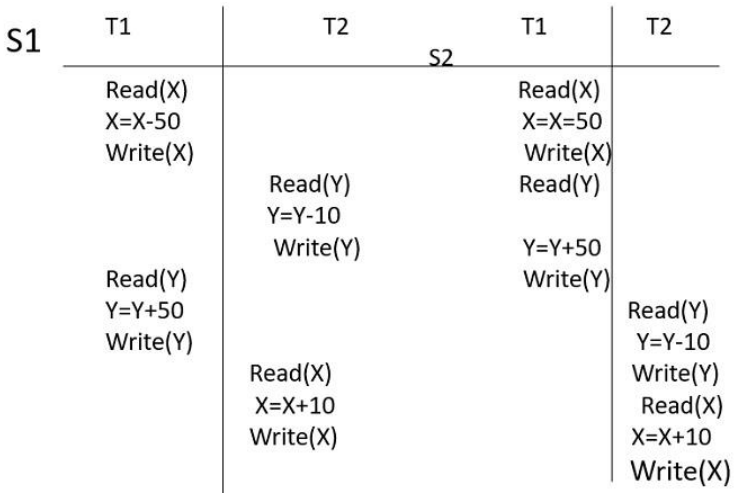
If there isn't any roll-back or the transaction comes from the "committed state", then the system is consistent and ready for new transaction and the old transaction is terminated.

4(b)

Two schedules are said to be conflict equivalent if the order of any two conflicting operations are the same in both the schedules.

Also, a concurrent schedule S is conflict equivalent to a serial schedule S', if we can obtain S' out of S by swapping the order of execution of non-conflicting instructions.

Example



Even if the schedule S1 keeps the database in the consistent state, we cannot convert it into a serial schedule and hence we conclude that the schedule is not conflict equivalent to any of the serial schedules.

So, instead of considering only the read and write operation, we will also consider the intermediate operations which will result in a new form of serializability known as view serializability.

So by seeing above transactions S1 is the conflict equivalent of S2.

Example 2

Two schedules are said to be conflict equivalent if the order of any two conflicting operations are the same in both the schedules.

The pre-requisite conditions for conflicting operations are −

- The two conflicting operations should belong to two different transactions.
- They should be acting over the same database or variable say x.

- At least one of the operations should be "Write". For example, operations should be like Read-write; Write-Write; Write-Read

Schedule 1	Schedule 2
R1(x)	R1(y)
W2(x)	R1(x)
R1(y)	W2(x)

Here in the example,

- R(x) - Read operation over variable x
- W(x) - Write operation over variable x
- R1W2 is the conflicting operation. As the order is the same in both the schedules.

Conflict Serializablity

As discussed in Concurrency control, serial schedules have less resource utilization and low throughput. To improve it, two or more transactions are run concurrently. But concurrency of transactions may lead to inconsistency in database. To avoid this, we need to check whether these concurrent schedules are serializable or not.

Conflict Serializable: A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

Conflicting operations: Two operations are said to be conflicting if all conditions satisfy:

- They belong to different transactions
- They operate on the same data item
- At Least one of them is a write operation

S1: R1(X), R1(Y), R2(X), R2(Y), W2(Y), W1(X)
S2: R1(X), R2(X), R2(Y), W2(Y), R1(Y), W1(X)

S1 is not conflict serializable, but S2 is conflict serializable

Conflict serializable means we can first run one Transaction then after it second without any conflict. Conflict occurs when data item is same and at least one operation is write.

So for S1, r1(X); r1(Y); r2(X); r2(Y); w2(Y); w1(X)

Schedule S1

```
  T1          T2
--------------------
 r1(X)
 r1(Y)
            r2(X)
            r2(Y)
            w2(Y)
 w1(X)
```

The schedule is neither conflict equivalent to T1T2, nor T2T1.

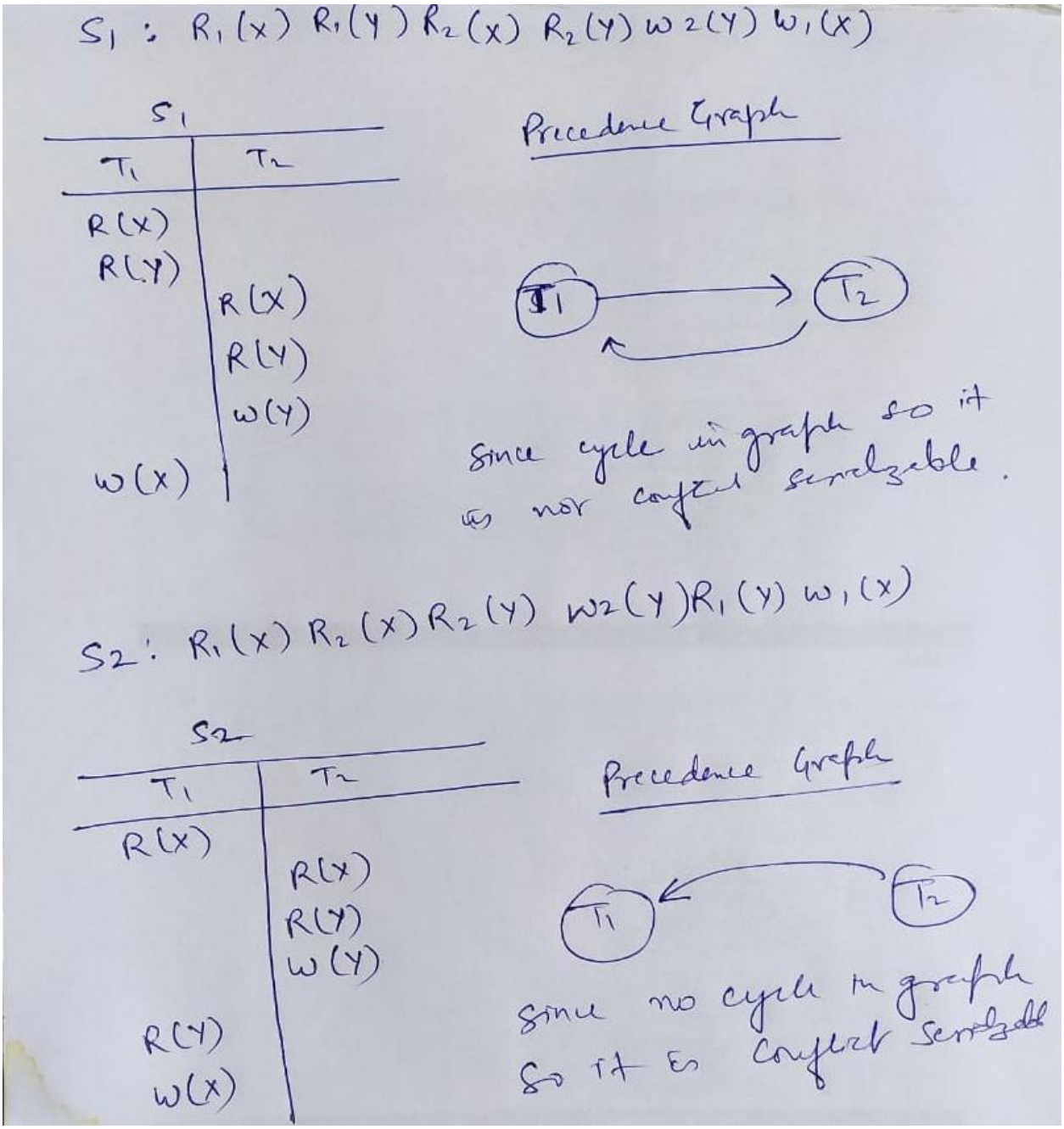
Now for S2, r1(X); r2(X); r2(Y); w2(Y); r1(Y); w1(X)

Schedule S2

T1	T2
r1(X)	
	r2(X)
	r2(Y)
	w2(Y)
r1(Y)	
w1(X)	

The schedule is conflict equivalent to T2T1.

Here r1(X) can be placed after w2(Y) as data item is different and before that only read operation are there so it is conflict serializable as T2 can be executed before T1



5 (a)

(i) What is Timestamp Ordering Protocol?

- Timestamp ordering protocol maintains the order of transaction based on their timestamps.
- A timestamp is a unique identifier that is being created by the DBMS when a transaction enters into the system. This timestamp can be based on the system clock or a logical counter maintained in the system.
- Timestamp helps identifying the older transactions (transactions that are waiting in line to be executed) and gives them higher priority compared to the newer transactions. This make sure that none of the transactions are pending for a longer period of time.
- This protocol also maintains the timestamps for the last read and last write on a data.
- For example, let's say an old transaction T1 timestamp is TS(T1) and a new transaction T2 enters into the system, timestamp assigned to T2 is TS(T2). Here TS(T1) < TS(T2) so the T1 has the higher priority because its timestamp is less than timestamp of T2. T1 would be given the higher priority than T2. This is how timestamp based protocol maintains the serializability order.

Problem:

Time	OP
1	S_1
2	$r_1(a)$
3	S_2
4	$r_2(b)$
5	$w_2(b)$
6	$w_1(a)$
7	S_3
8	$w_3(a)$
9	$w_3(b)$

T1 starts at TS =1

T2 starts at TS = 3

T3 starts at TS =7.

While giving the the TS for any read or write always look for youngest.

RTS(a) = a is first read by T1 hence RTS(a) =1. (Read(a) is never done anywhere again hence it is youngest)

WTS(a) = a is first written by T2 hence WTS(a) = 3. But again written by T3 which has higher TS (youngest) Hence final TS of W(a) = 7

RTS(b) = b is first read by T2 hence RTS(b) =3. (Read(b) is never done anywhere again hence it is youngest)

WTS(b) = b is first written by T2 hence WTS(b) = 3. But again written by T3 which has higher TS (youngest) Hence final TS of W(b) = 7

(ii) Timestamp-ordering concurrency control protocol with Thomas' Write Rule can generate view serializable schedules that are not conflict Serializable.
This statement is TRUE.

Timestamp ordering protocol guarantee conflict serializable schedules but say there exists schedules which are not conflict serializable due to blind writes for example,

$T1$	$T2$
$R(A)$	
	$W(A)$
$W(A)$	
$Committ$	
	$Commit$

Now this schedule is not conflict serializable and it also not allowed by basic time stamp protocol. But this schedule is view Serializable.

Now Thomas write rule says that if we ignore blind write conflict in time stamp based protocol then we will cover some view serializable schedules as well.

Timestamp-ordering concurrency control protocol with Thomas' Write Rule ignores outdated writes. Thomas Write Rue is a bit flexible, and allows more no. of schedules than Timestamp-ordering concurrency control protocol. **Hence it can generate View serial schedules for** those non-conflict serializable schedules with the testing of BLIND WRITE. Because of BLIND WRITE some of the non-conflict serializable schedules may be view-equivalent.

5 (b)

Multiple Granularity

It means hierarchically breaking up the database into blocks that can be locked. Multiple granularity increases concurrency and reduces the lock overhead. It makes it easy to decide which data should be locked and which is not. A tree can represent the hierarchy of multiple granularity.

For example: Consider a tree that has four levels of nodes.

- The root level or the top-level represents the entire database.
- The second level represents a node of the type area. The database consists of precisely these areas.
- The area has children nodes known as files. Any file cannot be present in more than one area.
- Each file has child nodes known as records. The file precisely consists of those records that are its child nodes. No records can be present in more than one file. Therefore, levels starting from the highest level are

1. Database
2. Area
3. File
4. Record

Explanation

- Consider the above diagram, each node in the tree can be locked individually. According to the 2-phase locking protocol, it will use shared and exclusive locks.

- When a transaction locks a node, it is either in shared or exclusive mode. The transaction implicitly locks descendants of that node in the same lock mode.
- Locks on files and records are made simple, but the lock on the root is still not determined.
- One way to know about a lock on the root is to search the entire tree. But it nullifies the whole concept of multiple granularity locking schemes.
- A more efficient way to achieve this is by using a new type of lock called **intention lock mode.**

Intention Lock Mode

In addition to shared and exclusive lock modes, other three lock modes are available.

1. **Intention-shared (IS):** explicit lock at a lower level of the tree but only with shared locks.
2. **Intention-Exclusive (IX):** explicit lock with exclusive or shared locks at a lower level.
3. **Shared & Intention-Exclusive (SIX):** In this lock, the node is locked in shared mode, and some nodes are locked in exclusive mode by the same transaction.

	S	X	IS	IX	IX
S	Yes	No	Yes	No	No
X	No	No	No	No	No
IS	Yes	No	Yes	Yes	No
SIX	No	No	Yes	No	No
IX	No	No	Yes	No	Yes

Intention lock modes are used in multiple-granularity locking protocol uses to ensure serializability. In this protocol, if a transaction T that attempts to lock a node need to follow these protocols:

1. Transaction Ti must follow the lock-compatibility matrix.
2. At first, Transaction Ti must lock the tree's root in any mode.
3. Transaction Ti will lock a node in S or IS mode only if Ti has locked that node's parent in either IS or IX mode.
4. Transaction Ti will lock a node in IX, SIX, or X mode only if Ti currently has the parent of the locked node in either SIX or IX modes.
5. Transaction Ti will lock a node only if Ti has not unlocked any node before (i.e., Ti is two-phase).
6. Transaction Ti will unlock a node only if Ti has not locked its children nodes currently.

Multiple granularity protocol requires that locks are made in a top-down (root-to-leaf) manner, whereas locks must be released in a bottom-up (leaf to-root) manner.

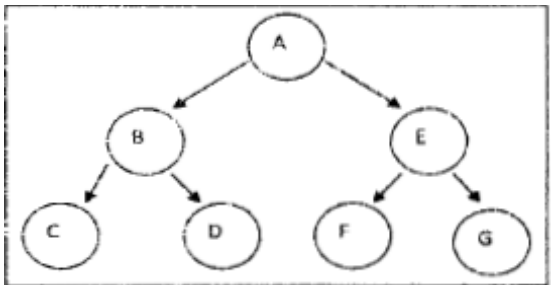
Consider the given tree in the figure and the transactions:

- Assume transaction T1 reads record Ra2 from file Fa. Then, T2 will lock the database, area A1, and Fa in IS mode (and in that order), and finally, it will lock Ra2 in S mode.
- Assume transaction T2 modifies record Ra9 from file Fa. Then, T2 will lock the database, area A1, and file Fa (in that order) in IX mode and lock Ra9 in X mode.

- Assume transaction T3 reads all the records from file Fa. Then, T3 will lock the database and area A1 (and in that order) in IS mode and lock Fa in S mode.
- Assume transaction T4 reads the entire database. It can be done after locking the database in S mode.

Transactions T1, T3, and T4 can access the database together. Transaction T2 can execute concurrently with T1 but not with T3 or T4.

Deadlock is also possible in the multiple-granularity protocol, as it is in the two-phase locking protocol. These can be eliminated by using specific deadlock elimination techniques.



T1 wants to access item C in read mode.
We need to impose lock on C in shared mode:
IS-Lock(A) -> IS-Lock(B) -> S-Lock(C)
T2 wants to access item D in exclusive mode.
We need to impose lock on D in exclusive mode:
IX-Lock(A) -> IX-Lock(B) -> X-Lock(D)
T3 wants to read all the children of item B.
We need to impose lock on B in shared mode, but shared lock is not compatible with intension exclusive mode, so shared lock will not be granted on B and hence transaction is rollback.
T4 wants to access A in read mode.
We need to impose lock on A in shared mode, but shared lock is not compatible with intension exclusive mode, so shared lock will not be granted on A and hence transaction is rollback.