CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | |

Internal Assessment Test II – August 2023

Sub:	**Database Management System**						**Sub Code:**	**22MCA21**	
Date:	**28-08-2023**	**Duration:**	**90 min's**	**Max Marks:**	**50**	**Sem:**	**II**	**Branch:**	**MCA**

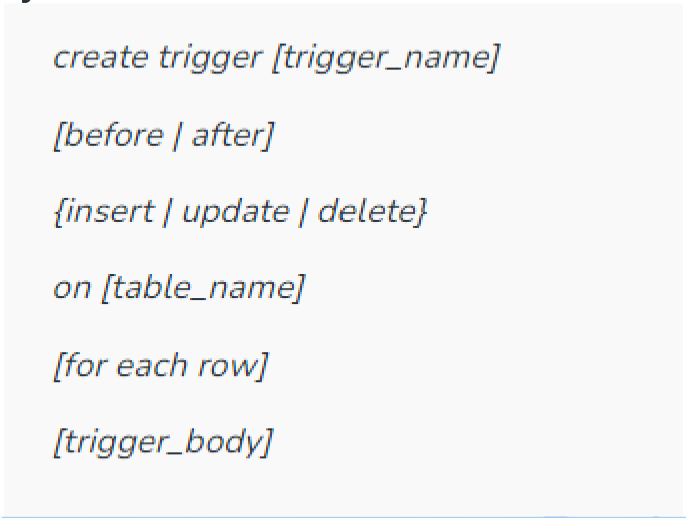
Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

		MARKS	OBE	
	PART I		CO	RBT
1	What are triggers in SQL? Explain about Triggers in SQL with Suitable example. **OR**	[10]	CO2	L3
2	What are triggers in SQL? Explain about Triggers in SQL with Suitable example.	[10]	CO3	L2
3	**PART II** What are the different types of JOINs? Provide examples for each by explaining the scenarios where each type of join is commonly used. **OR**	[10]	CO2	L3
4	What are Inference Rules? Explain any 6 IR rules with examples.	[10]	CO3	L2

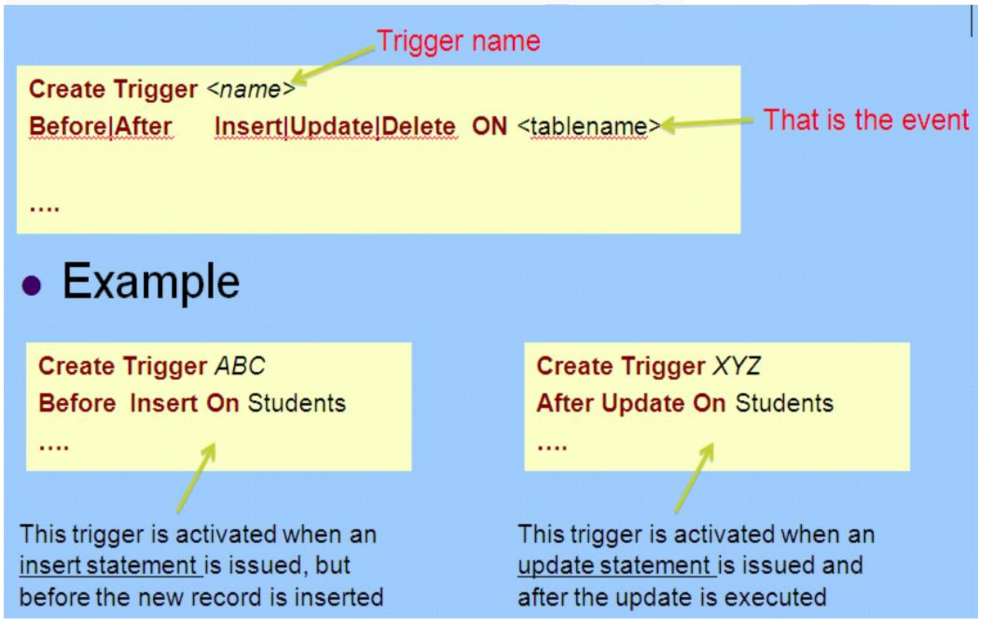
5	**PART III** Define Normalization. Explain 1NF, 2NF and 3NF with examples. **OR**	[10]	CO4	L3
6	What are Views in SQL? Discuss on methodologies to implement views in SQL. Explain with an example.	[10]	CO4	L2
7	PART IV How assertions are used to enforce complex constraint? Explain with example. **OR**	[10]	CO2	L2
8	Given the functional dependencies X= {A -> B, AB->C, D->AC, D->E} and Y= {A -> BC, D -> AE}, Explain whether these two sets of functional dependencies are equivalent.	[10]	CO3	L3
9	PART V In the context of Embedded SQL, what is a cursor? How is it used, and what problem does it help to solve? **OR**	[10]	CO3	L2
10	Let the given set of Functional Dependencies be X: {B->A, A->D, AB->D}. Find the minimal cover of X.	[10]	CO3	L3

Solution

1. **What are triggers in SQL? Explain about Triggers in SQL with Suitable example.**
   Another important statement in SQL is CREATE TRIGGER. In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied. For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation. The CREATE TRIGGER statement is used to implement such actions in SQL. A typical trigger has three components:
   Event: When this event happens, the trigger is activated.
   Condition (optional): If the condition is true, the trigger executes, otherwise skipped
   Action: The action performed by the trigger
   The action is to be executed automatically if the condition is satisfied when event occurs.¬
   Trigger: Events Three event types Insert Update Delete Two triggering times Before the event After the event Two granularities Execute for each row Execute for each statement
   Syntax:



```
create trigger [trigger_name]

[before | after]

{insert | update | delete}

on [table_name]

[for each row]

[trigger_body]
```



```
Create Trigger <name>
Before|After     Insert|Update|Delete  ON <tablename>

....
```

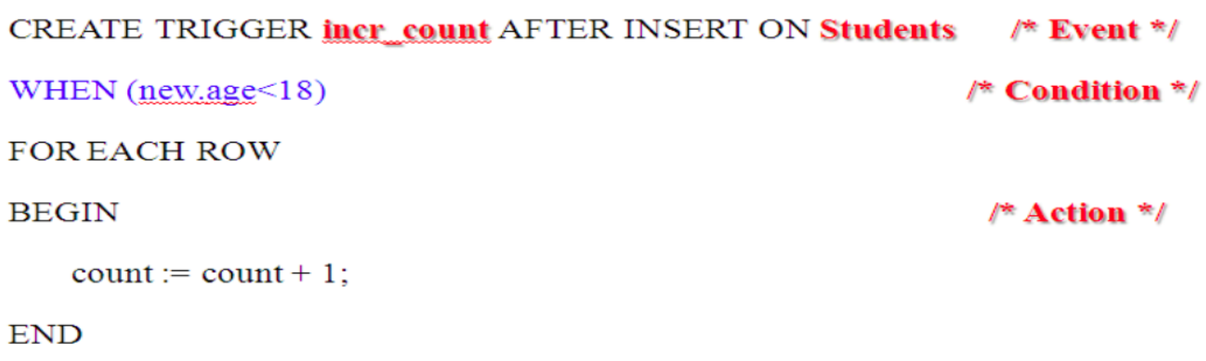
• Example

```
Create Trigger ABC
Before  Insert On Students
....
```

This trigger is activated when an insert statement is issued, but before the new record is inserted

```
Create Trigger XYZ
After Update On Students
....
```

This trigger is activated when an update statement is issued and after the update is executed



```
CREATE TRIGGER incr_count AFTER INSERT ON Students      /* Event */
WHEN (new.age<18)                                       /* Condition */
FOR EACH ROW
BEGIN                                                   /* Action */
    count := count + 1;
END
```

For example, given Library Book Management database schema with Student database schema. In these databases, if any student borrows a book from library then the count of that specified book should be decremented. To do so,

Suppose the schema with some data,

```
mysql> select * from book_det;
+-----+-------------+--------+
| bid | btitle      | copies |
+-----+-------------+--------+
|   1 | Java        |     10 |
|   2 | C++         |      5 |
|   3 | MySql       |     10 |
|   4 | Oracle DBMS |      5 |
+-----+-------------+--------+
4 rows in set (0.00 sec)

mysql> select * from book_issue;
+------+------+--------+
| bid  | sid  | btitle |
+------+------+--------+
1 row in set (0.00 sec)
```

To implement such procedure, in which if the system inserts the data into the book_issue database a trigger should automatically invoke and decrements the copies attribute by 1 so that a proper track of book can be maintained.

```
create trigger book_copies_deducts
after INSERT
on book_issue
for each row
update book_det set copies = copies - 1 where bid = new.bid;
```

Above trigger, will be activated whenever an insertion operation performed in a book_issue database, it will update the book_det schema setting copies decrements by 1 of current book id(bid).

2. Explain the informal design guidelines for relation schema

- Making sure that the semantics of the attributes is clear in the schema
- Reducing the redundant information in tuples
- Reducing the NULL values in tuples
- Disallowing the possibility of generating spurious tuples

1. Semantics of the Attributes

Whenever we are going to form relational schema there should be some meaning among the attributes.This meaning is called semantics.This semantics relates one attribute to another with some relation.

Eg:

USN No

________	Student name	Sem

2. Reducing the Redundant Value in Tuples

Mixing attributes of multiple entities may cause problems

Information is stored redundantly wasting storage

Problems with update anomalies

Insertion anomalies

Deletion anomalies

Modification anomalies

______	Student name	Sem

Eg:

Dept No	Dept Name

If we integrate these two and is used as a single table i.e Student Table

USN No	Student name	Sem	Dept No	Dept Name

Here whenever if we insert the tuples there may be 'N' stunents in one department,so Dept No,Dept Name values are repeated 'N' times which leads to data redundancy.

Another problem is updata anamolies ie if we insert new dept that has no students.

If we delet the last student of a dept,then whole information about that department will be deleted

If we change the value of one of the attributes of aparticaular table the we must update the tuples of all the students belonging to thet depy else Database will become inconsistent.

Note: Design in such a way that no insertion ,deletion,modification anamolies will occur

3. Reducing Null values in Tuples.

Note: Relations should be designed such that their tuples will have as few NULL values as possible

Attributes that are NULL frequently could be placed in separate relations (with the primary key)

Reasons for nulls:

attribute not applicable or invalid

attribute value unknown (may exist)

value known to exist, but unavailable

4. Disallowing spurious Tuples

Bad designs for a relational database may result in erroneous results for certain JOIN operations

The "lossless join" property is used to guarantee meaningful results for join operations

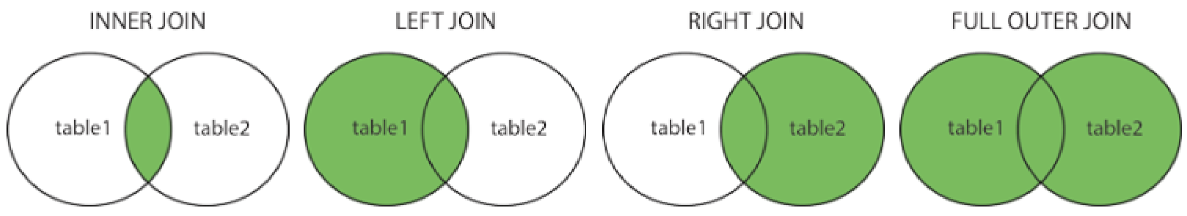
Note: The relations should be designed to satisfy the lossless join condition. No spurious tuples should be generated by doing a natural-join of any relations.

3. **What are the different types of JOINs? Provide examples for each by explaining the scenarios where each type of join is commonly used.**
A `JOIN` clause is used to combine rows from two or more tables, based on a related column between them.

Different Types of SQL JOINs

Here are the different types of the JOINs in SQL:

- `(INNER) JOIN`: Returns records that have matching values in both tables
- `LEFT (OUTER) JOIN`: Returns all records from the left table, and the matched records from the right table
- `RIGHT (OUTER) JOIN`: Returns all records from the right table, and the matched records from the left table
- `FULL (OUTER) JOIN`: Returns all records when there is a match in either left or right table



4. What are Inference Rules? Explain any 6 IR rules with examples.

Inference Rule (IR):

- The Armstrong's axioms are the basic inference rule.
- Armstrong's axioms are used to conclude functional dependencies on a relational database.
- The inference rule is a type of assertion. It can apply to a set of FD(functional dependency) to derive other FD.
- Using the inference rule, we can derive additional functional dependency from the initial set.

The Functional dependency has 6 types of inference rule:

1. Reflexive Rule (IR_1)

In the reflexive rule, if Y is a subset of X, then X determines Y.

```
If X ⊇ Y then X  →   Y
```

Example:

```
X = {a, b, c, d, e}
Y = {a, b, c}
```

2. Augmentation Rule (IR_2)

The augmentation is also called as a partial dependency. In augmentation, if X determines Y, then XZ determines YZ for any Z.

```
If X   → Y then XZ  →  YZ
```

Example:

```
For R(ABCD),  if A  →  B then AC  →  BC
```

3. Transitive Rule (IR_3)

In the transitive rule, if X determines Y and Y determine Z, then X must also determine Z.

```
If X  →  Y and Y → Z then X →  Z
```

4. Union Rule (IR_4)

Union rule says, if X determines Y and X determines Z, then X must also determine Y and Z.

```
If X   → Y and X  → Z then X →   YZ
```

Proof:

1. $X \rightarrow Y$ (given)
2. $X \rightarrow Z$ (given)
3. $X \rightarrow XY$ (using IR_2 on 1 by augmentation with X. Where XX = X)
4. $XY \rightarrow YZ$ (using IR_2 on 2 by augmentation with Y)
5. $X \rightarrow YZ$ (using IR_3 on 3 and 4)

5. Decomposition Rule (IR_5)

Decomposition rule is also known as project rule. It is the reverse of union rule.

This Rule says, if X determines Y and Z, then X determines Y and X determines Z separately.

If $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$

Proof:

1. $X \rightarrow YZ$ (given)
2. $YZ \rightarrow Y$ (using IR_1 Rule)
3. $X \rightarrow Y$ (using IR_3 on 1 and 2)

6. Pseudo transitive Rule (IR_6)

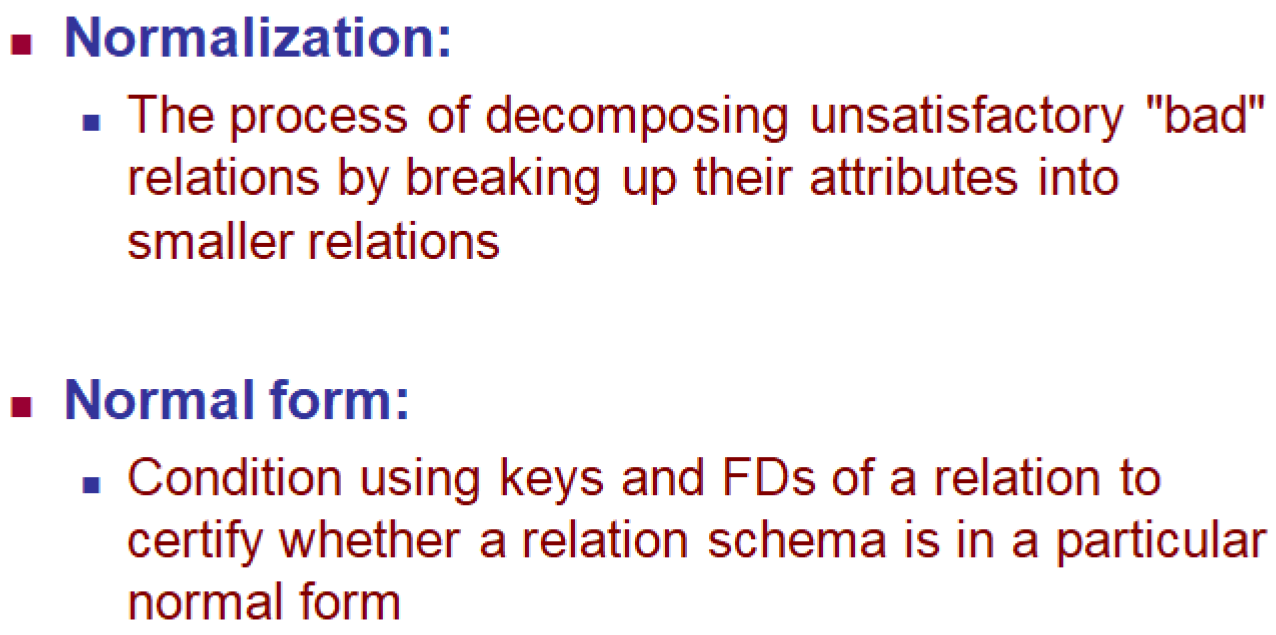
In Pseudo transitive Rule, if X determines Y and YZ determines W, then XZ determines W.

If $X \rightarrow Y$ and $YZ \rightarrow W$ then $XZ \rightarrow W$

Proof:

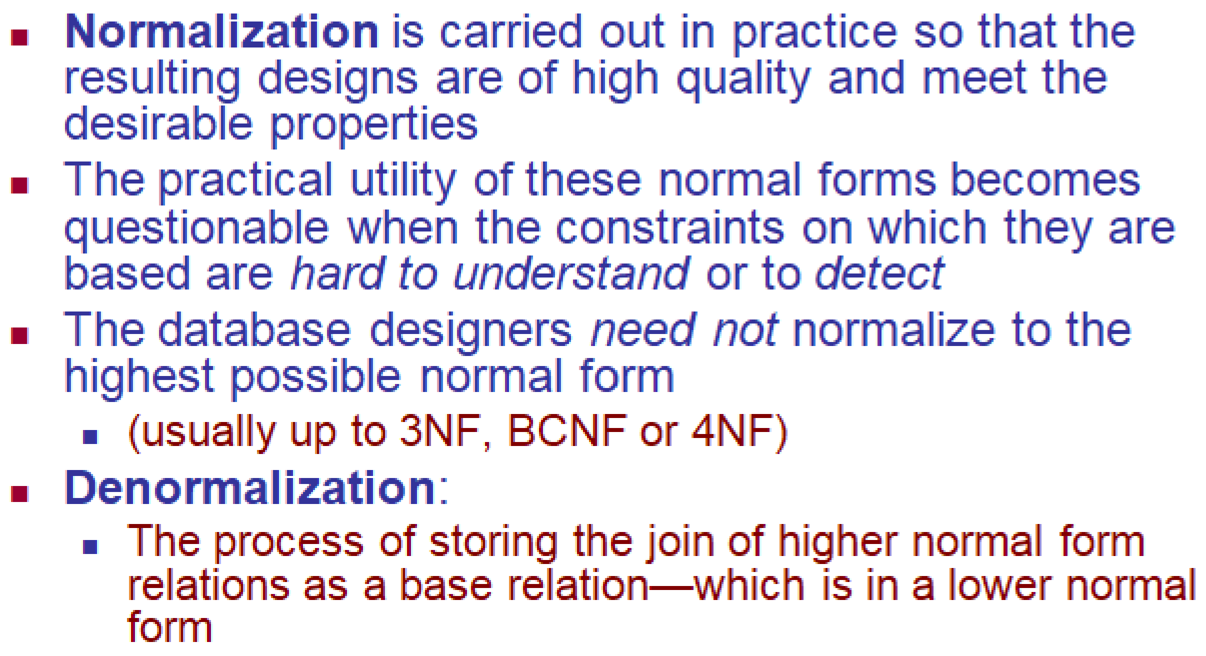
1. $X \rightarrow Y$ (given)
2. $WY \rightarrow Z$ (given)
3. $WX \rightarrow WY$ (using IR_2 on 1 by augmenting with W)
4. $WX \rightarrow Z$ (using IR_3 on 3 and 2)

5. Define Normalization. Explain 1NF, 2NF and 3NF with examples.



- **Normalization:**
  - The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations

- **Normal form:**
  - Condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form



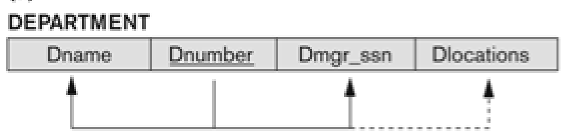
- **Normalization** is carried out in practice so that the resulting designs are of high quality and meet the desirable properties
- The practical utility of these normal forms becomes questionable when the constraints on which they are based are *hard to understand* or to *detect*
- The database designers *need not* normalize to the highest possible normal form
  - (usually up to 3NF, BCNF or 4NF)
- **Denormalization**:
  - The process of storing the join of higher normal form relations as a base relation—which is in a lower normal form

First Normal Form

- Disallows
  - composite attributes
  - multivalued attributes
  - **nested relations**; attributes whose values for an *individual tuple* are non-atomic

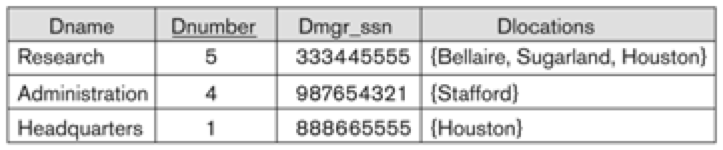
- Considered to be part of the definition of relation

(a)



(b)

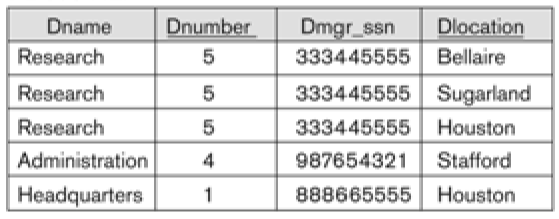
DEPARTMENT



Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT



Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Second Normal Form

- Uses the concepts of **FDs, primary key**
- Definitions
  - **Prime attribute:** An attribute that is member of the primary key K
  - **Full functional dependency:** a FD Y -> Z where removal of any attribute from Y means the FD does not hold any more
- Examples:
  - {SSN, PNUMBER} -> HOURS is a full FD since neither SSN -> HOURS nor PNUMBER -> HOURS hold
  - {SSN, PNUMBER} -> ENAME is not a full FD (it is called a partial dependency) since SSN -> ENAME also holds

- A relation schema R is in **second normal form (2NF)** if every non-prime attribute A in R is fully functionally dependent on the primary key

- R can be decomposed into 2NF relations via the process of 2NF normalization

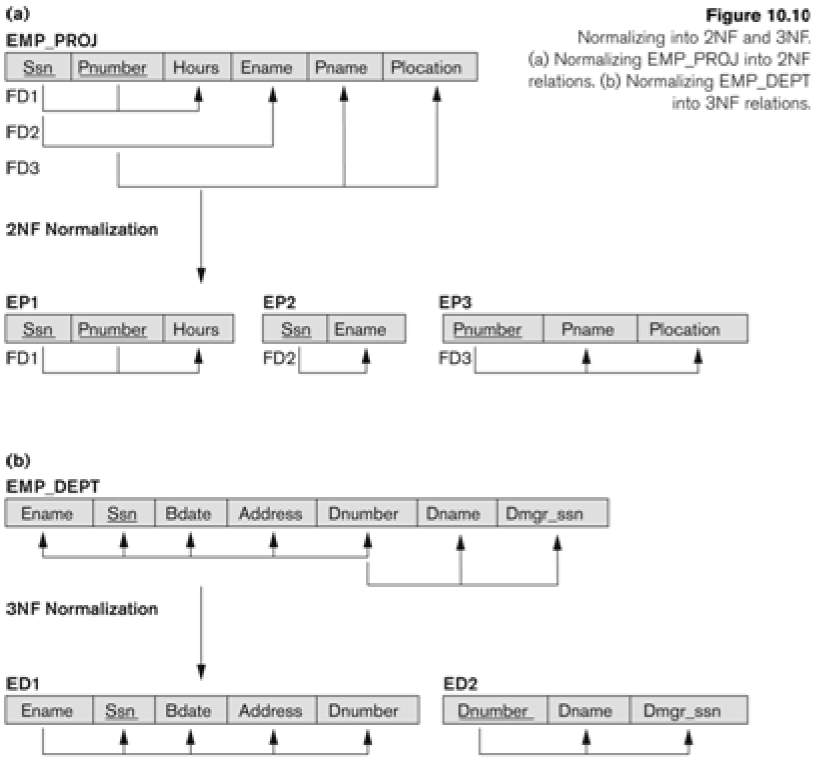


Figure 10.10
Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

3.4 Third Normal Form (1)

- Definition:
  - **Transitive functional dependency:** a FD X -> Z that can be derived from two FDs X -> Y and Y -> Z
- Examples:
  - SSN -> DMGRSSN is a **transitive** FD
    - Since SSN -> DNUMBER and DNUMBER -> DMGRSSN hold
  - SSN -> ENAME is **non-transitive**
    - Since there is no set of attributes X where SSN -> X and X -> ENAME

6. **What are Views in SQL? Discuss on methodologies to implement views in SQL. Explain with an example.**

A view is a single table that is derived from one or more base tables or other views .Views neither exist physically nor contain data itself, it depends on the base tables for its existence A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

Specification of Views in SQL

Syntax:

CREATE VIEW view_name AS SELECT column_name(s) FROM table_name WHERE condition

Example

CREATE VIEW WORKS_ON1 AS SELECT Fname, Lname, Pname, Hours FROM EMPLOYEE, PROJECT, WORKS_ON WHERE Ssn=Essn ANDPno=Pnumber ;

Retrieve the Last name and First name of all employees who work on ‘ProductX’

SELECT Fname, Lname FROM WORKS_ON1 WHERE Pname=‘ProductX’ ;

A view always shows up-to-date¬ If we modify the tuples in the base tables on which the view is defined, the view must automatically¬ reflect these changes If we do not need a view any more, we can use the DROP VIEW command¬ DROP VIEW WORKS_ON1;

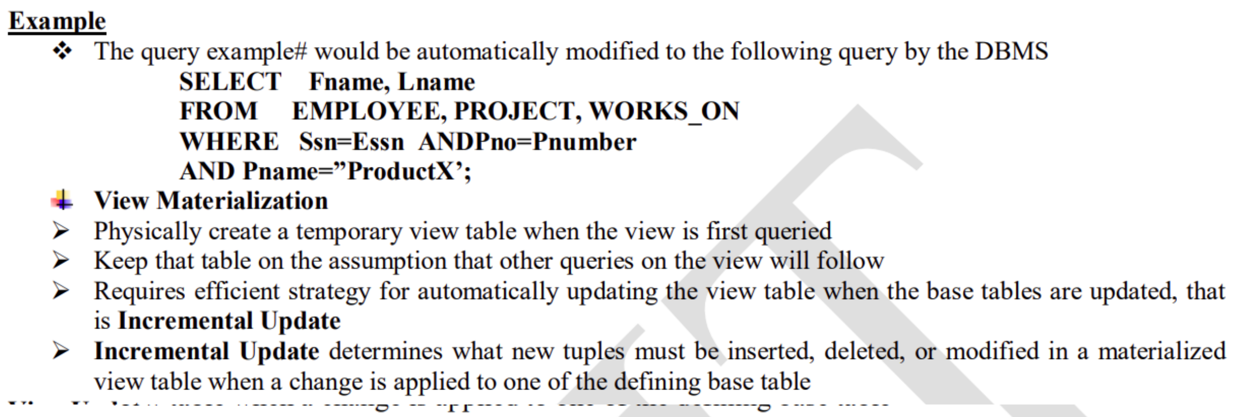
View Implementation and View Update

View Implementation

The problem of efficiently implementing a view for quering is complex two main approaches have been suggested

Modifying the view query into a query on the underlying base tables

Disadvantage: inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple queries are applied to the view within a short period of time.



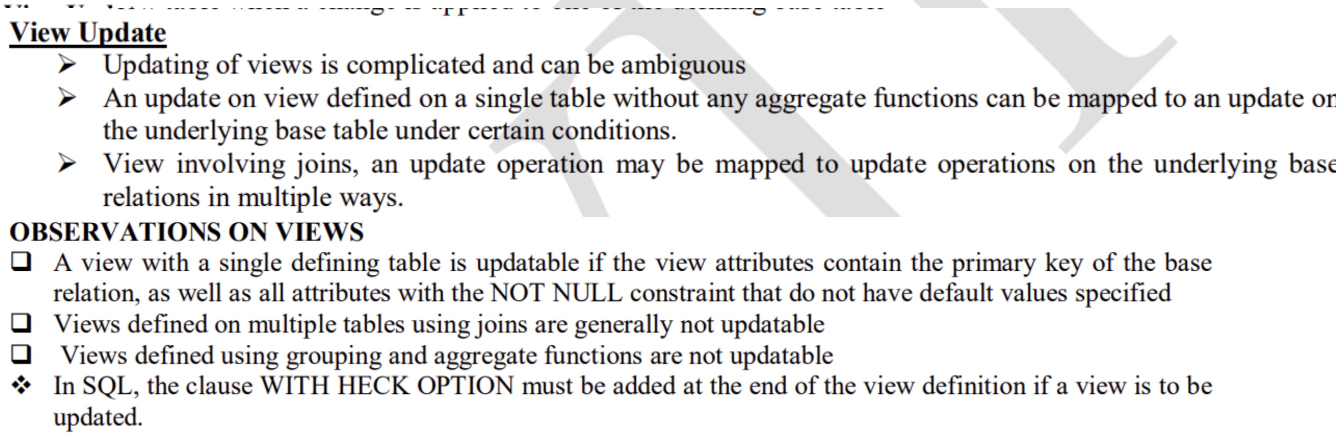
Example

❖ The query example# would be automatically modified to the following query by the DBMS

```
SELECT   Fname, Lname
FROM     EMPLOYEE, PROJECT, WORKS_ON
WHERE    Ssn=Essn  ANDPno=Pnumber
AND Pname=”ProductX’;
```

View Materialization

- Physically create a temporary view table when the view is first queried
- Keep that table on the assumption that other queries on the view will follow
- Requires efficient strategy for automatically updating the view table when the base tables are updated, that is **Incremental Update**
- **Incremental Update** determines what new tuples must be inserted, deleted, or modified in a materialized view table when a change is applied to one of the defining base table



View Update

- Updating of views is complicated and can be ambiguous
- An update on view defined on a single table without any aggregate functions can be mapped to an update o the underlying base table under certain conditions.
- View involving joins, an update operation may be mapped to update operations on the underlying base relations in multiple ways.

OBSERVATIONS ON VIEWS

- ❑ A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint that do not have default values specified
- ❑ Views defined on multiple tables using joins are generally not updatable
- ❑ Views defined using grouping and aggregate functions are not updatable
- ❖ In SQL, the clause WITH HECK OPTION must be added at the end of the view definition if a view is to be updated.

Advantages of Views

- Data independence
- Currency
- Improved security
- Reduced complexity
- Convenience
- Customization
- Data integrity

7. How assertions are used to enforce complex constraint? Explain with example.

Specifying Constraints as Assertions and Actions as Triggers

CREATE ASSERTION, which can be used to specify additional types of constraints that are outside the scope of the built-in relational model constraints (primary and unique keys, entity integrity, and referential integrity) that we presented early.

CREATE TRIGGER,which can be used to specify automatic actions that the database system will perform when certain events and conditions occur. This type of functionality is generally referred to as active databases.

In SQL,users can specify general constraints—those that do not fall into any of the categories described via declarative assertions, using the CREATE ASSERTION statement of the DDL. Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.

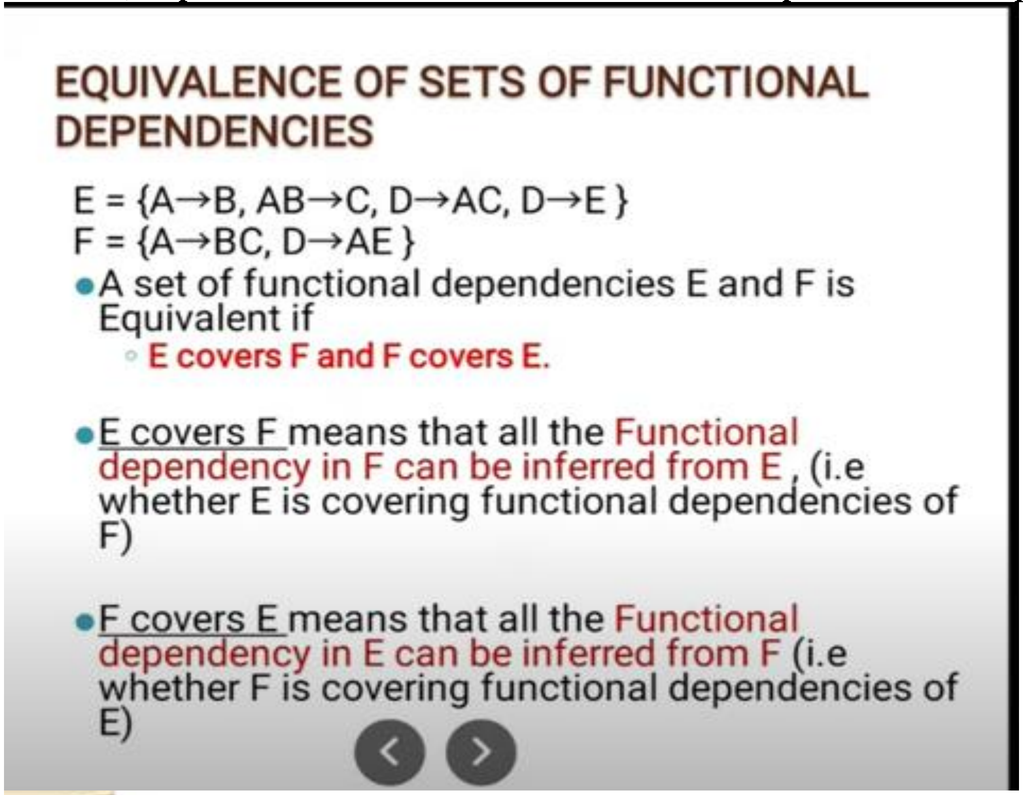
For example, to specify the constraint that *the salary of an employee must not be greater than the salary of the manager of the department that the employee works for* in SQL, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS  ( SELECT    *
                      FROM      EMPLOYEE E, EMPLOYEE M,
                                DEPARTMENT D
                      WHERE     E.Salary>M.Salary
                                AND E.Dno=D.Dnumber
                                AND D.Mgr_ssn=M.Ssn ) );
```

The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK, which is followed by a **condition** in parentheses that must hold true on every database state for the assertion to be satisfied. The constraint name can be used later to refer to the constraint or to modify or drop it. The DBMS is responsible for ensuring that the condition is not violated. Any WHERE clause condition can be used, but many constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions. Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is **violated**. The constraint is **satisfied** by a database state if *no combination of tuples* in that database state violates the constraint.

The basic technique for writing such assertions is to specify a query that selects any tuples that violate the desired condition.By including this query inside a NOT EXISTSclause, the assertion will specify that the result of this query must be empty so that the condition will always be TRUE.Thus,the assertion is violated if the result of the query is not empty.In the preceding example,the query selects all employees whose salaries are greater than the salary of the manager of their department. If the result of the query is not empty,the assertion is violated.

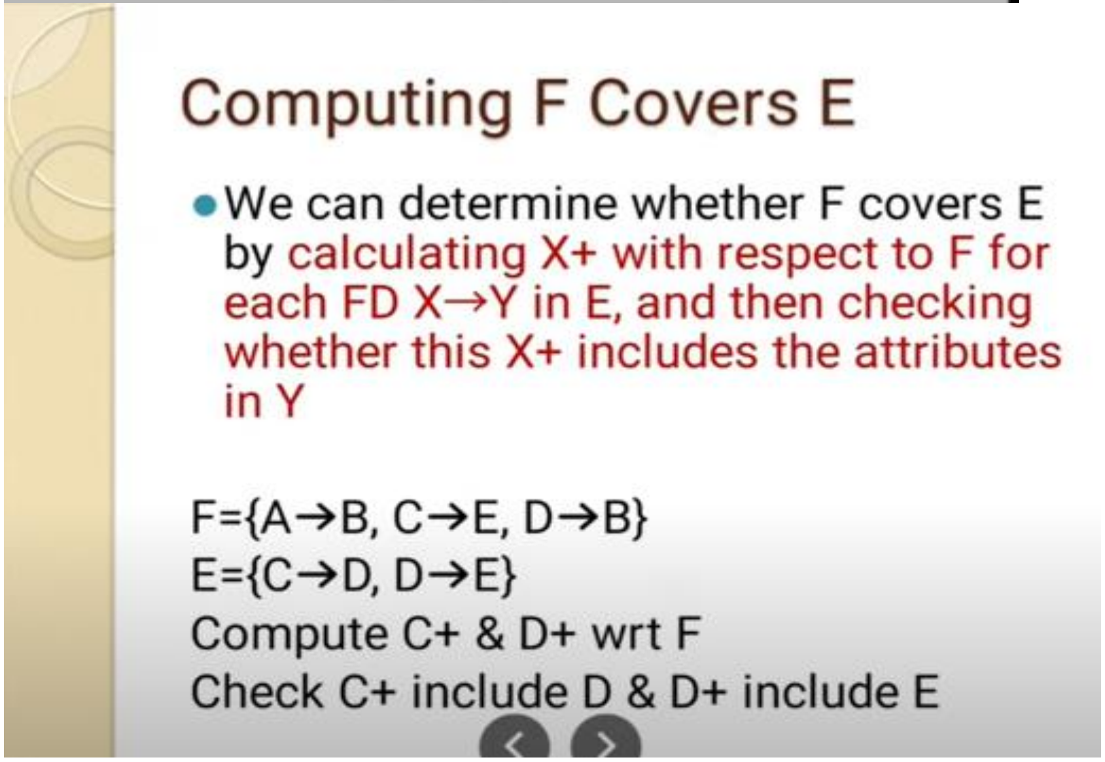
8. Given the functional dependencies X= {A -> B, AB->C, D->AC, D->E} and Y= {A -> BC, D -> AE}, Explain whether these two sets of functional dependencies are equivalent.



EQUIVALENCE OF SETS OF FUNCTIONAL DEPENDENCIES

E = {A→B, AB→C, D→AC, D→E }
F = {A→BC, D→AE }

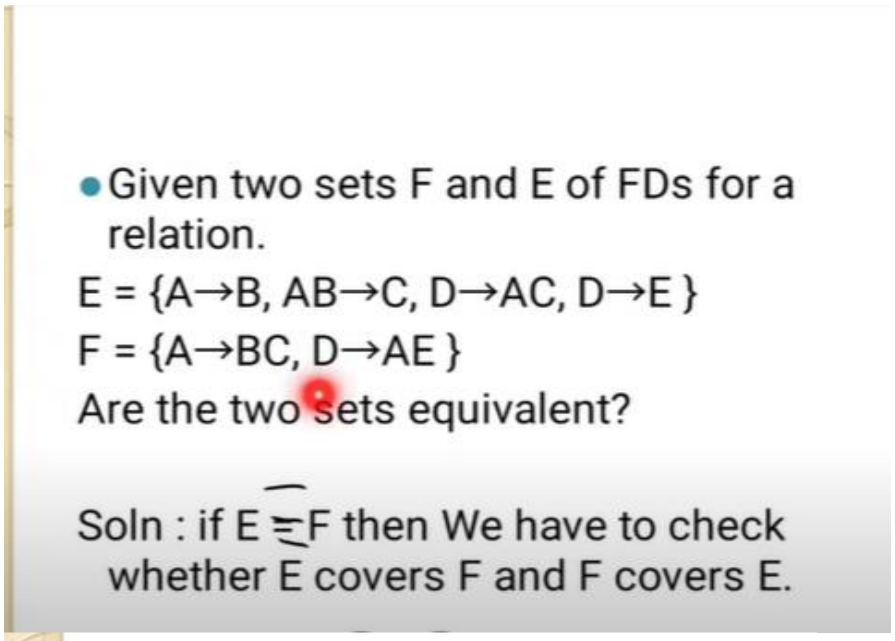
- A set of functional dependencies E and F is Equivalent if
  - E covers F and F covers E.
- E covers F means that all the Functional dependency in F can be inferred from E , (i.e whether E is covering functional dependencies of F)
- F covers E means that all the Functional dependency in E can be inferred from F (i.e whether F is covering functional dependencies of E)



Computing F Covers E

- We can determine whether F covers E by calculating X+ with respect to F for each FD X→Y in E, and then checking whether this X+ includes the attributes in Y

F={A→B, C→E, D→B}
E={C→D, D→E}
Compute C+ & D+ wrt F
Check C+ include D & D+ include E



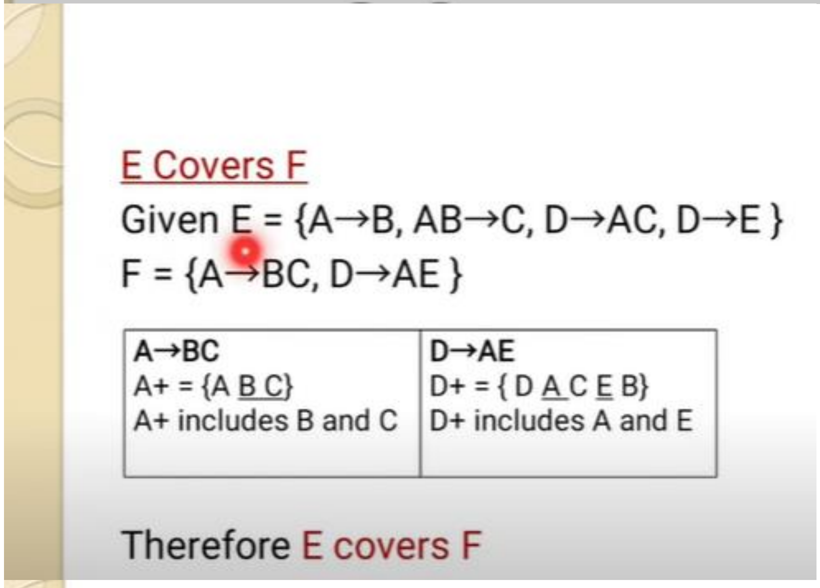
- Given two sets F and E of FDs for a relation.

E = {A→B, AB→C, D→AC, D→E }

F = {A→BC, D→AE }

Are the two sets equivalent?

Soln : if $E \equiv F$ then We have to check whether E covers F and F covers E.



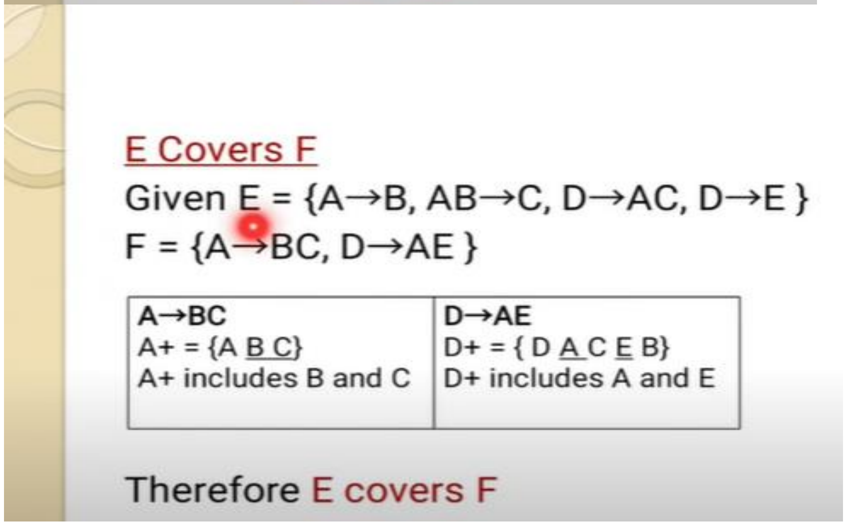
E Covers F

Given E = {A→B, AB→C, D→AC, D→E }

F = {A→BC, D→AE }

A→BC	D→AE
A+ = {A $\underline{B}$ $\underline{C}$}	D+ = { D $\underline{A}$ C $\underline{E}$ B}
A+ includes B and C	D+ includes A and E

Therefore E covers F



E Covers F

Given E = {A→B, AB→C, D→AC, D→E }

F = {A→BC, D→AE }

A→BC	D→AE
A+ = {A $\underline{B}$ $\underline{C}$}	D+ = { D $\underline{A}$ C $\underline{E}$ B}
A+ includes B and C	D+ includes A and E

Therefore E covers F

9. **In the context of Embedded SQL, what is a cursor? How is it used, and what problem does it help to solve?**

Cursors:

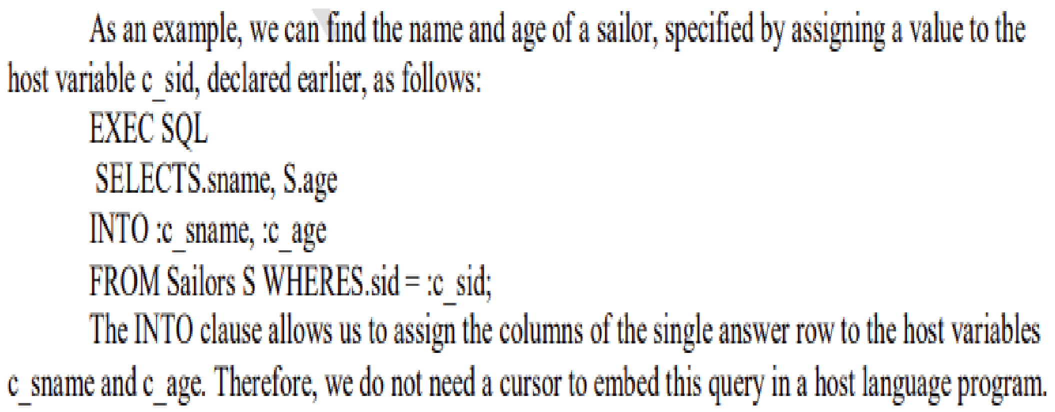
A major problem in embedding SQL statements in a host language like C is that an impedance mismatch occurs because SQL operates on set of records, whereas languages like C do not cleanly support a set-of-records abstraction. The solution is to essentially provide a mechanism that allows us to retrieve rows one at a time from a relation.This mechanism is called a cursor. We can declare a cursor on any relation or on any SQL query (because every query returns a set of rows). Once a cursor is declared, we can open it (which positions the cursor just before the first row); fetch the next row; move the cursor (to the next row, to the row after the next n, to the first row, or to the previous row, etc., by specifying additional parameters for the FETCH command); or close the cursor. Thus, a cursor essentially allows us to retrieve the rows in a table by positioning the cursor at a particular row and reading its contents.

Basic Cursor Definition and Usage

cursors enable us to examine, in the host language program, a collection of rows computed by an Embedded SQL statement:

We usually need to open a cursor if the embedded statement is a SELECT query. However, we can avoid opening a cursor if the answer contains a single row.

INSERT, DELETE, and UPDATE statements typically require no cursor, although some variants of DELETE and UPDATE use a cursor.



As an example, we can find the name and age of a sailor, specified by assigning a value to the host variable c_sid, declared earlier, as follows:

```
EXEC SQL
 SELECTS.sname, S.age
INTO :c_sname, :c_age
FROM Sailors S WHERES.sid = :c_sid;
```

The INTO clause allows us to assign the columns of the single answer row to the host variables c_sname and c_age. Therefore, we do not need a cursor to embed this query in a host language program.

But what about the following query, which computes the names and ages of all sailors with a rating greater than the current value of the host variable c_minrating?

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.rating> :c_minrating
```

This query returns a collection of rows, not just one row. 'When executed interactively, the answers are printed on the screen. If we embed this query in a C program by prefixing the cOlnmand with EXEC SQL, how can the answers be bound to host language variables? The INTO clause is inadequate because we must deal with several rows. The solution is to use a cursor:

```
DECLARE sinfo CURSOR FOR
SELECT  S.sname, S.age
FROM    Sailors S
WHERE   S.rating > :c_minrating;
```

This code can be included in a C program, and once it is executed, the cursor *sinfo* is defined. Subsequently, we can open the cursor:

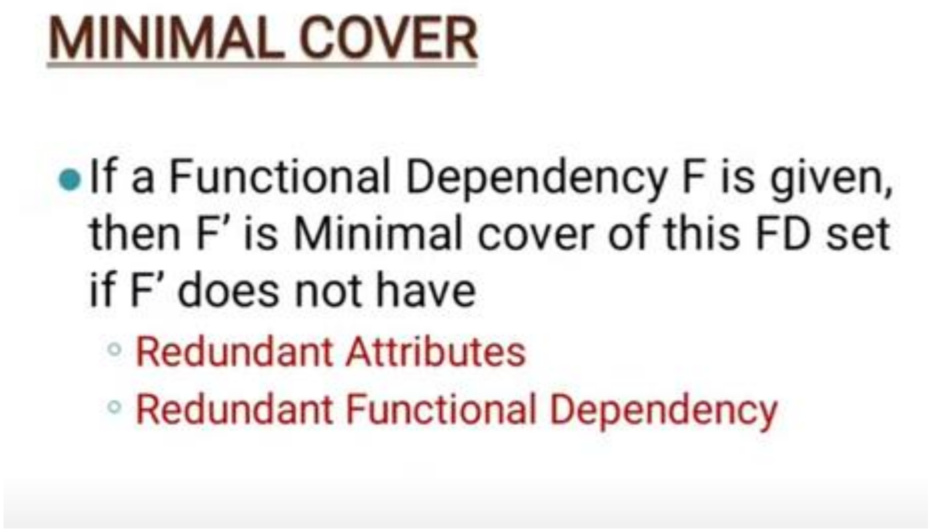
```
OPEN sinfo:
```

The value of *c_minrating* in the SQL query associated with the cursor is the value of this variable when we open the cursor. (The cursor declaration is processed at compile-time, and the OPEN command is executed at run-time.) A cursor can be thought of as 'pointing' to a row in the collection of answers to the query associated with it. When a cursor is opened, it is positioned just before the first row. We can use the FETCH command to read the first row of cursor sinfo into host language variables:

FETCH sinfoINTO :c_sname, :c_age;

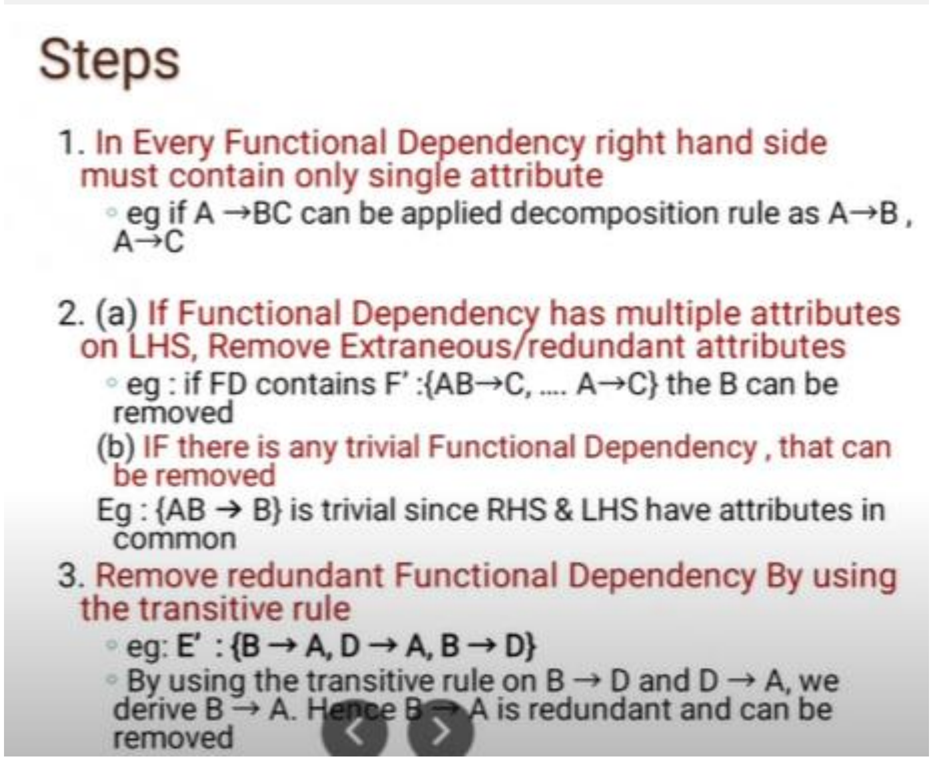
When the FETCH statement is executed, the cursor is positioned to point at the next row (which is the first row in the table when FETCH is executed for the first time after opening the cursor) and the column values in the row are copied into the corresponding host variables. By repeatedly executing this FETCH statement (say, in a while-loop in the C program), we can read all the rows computed by the query, one row at a time. Additional parameters to the FETCH command allow us to position a cursor in very flexible ways. How do we know when we have looked at all the rows associated with the cursor? By looking at the special variables SQLCODE or SQLSTATE, of course. SQLSTATE, for example, is set to the value 02000, which denotes NO DATA, to indicate that there are no more rows ifthe FETCH statement positions the cursor after the last row. When we are done with a cursor, we can close it: **CLOSE sinfo;** It can be opened again if needed and the value of :c_minrating in the SQL query associated with the cursor would be the value of the host variable c_minrating at that time

10. **Let the given set of Functional Dependencies be X: {B->A, A->D, AB->D}. Find the minimal cover of X.**



MINIMAL COVER

- If a Functional Dependency F is given, then F' is Minimal cover of this FD set if F' does not have
  - Redundant Attributes
  - Redundant Functional Dependency

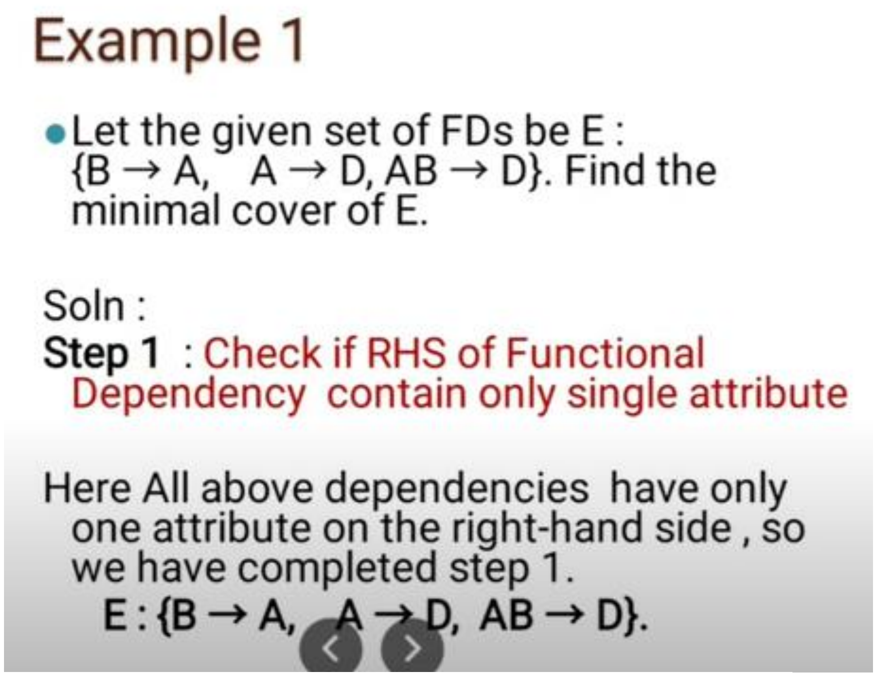


Steps

1. In Every Functional Dependency right hand side must contain only single attribute
   - eg if A →BC can be applied decomposition rule as A→B , A→C
2. (a) If Functional Dependency has multiple attributes on LHS, Remove Extraneous/redundant attributes
   - eg : if FD contains F' :{AB→C, ... A→C} the B can be removed

   (b) IF there is any trivial Functional Dependency , that can be removed

   Eg : {AB → B} is trivial since RHS & LHS have attributes in common
3. Remove redundant Functional Dependency By using the transitive rule
   - eg: E' : {B → A, D → A, B → D}
   - By using the transitive rule on B → D and D → A, we derive B → A. Hence B → A is redundant and can be removed



Example 1

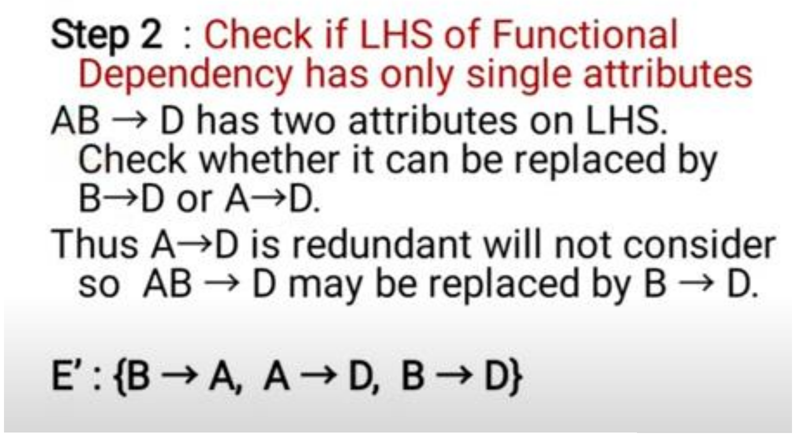
- Let the given set of FDs be E : {B → A, A → D, AB → D}. Find the minimal cover of E.

Soln :

Step 1 : Check if RHS of Functional Dependency contain only single attribute

Here All above dependencies have only one attribute on the right-hand side , so we have completed step 1.

E : {B → A, A → D, AB → D}.

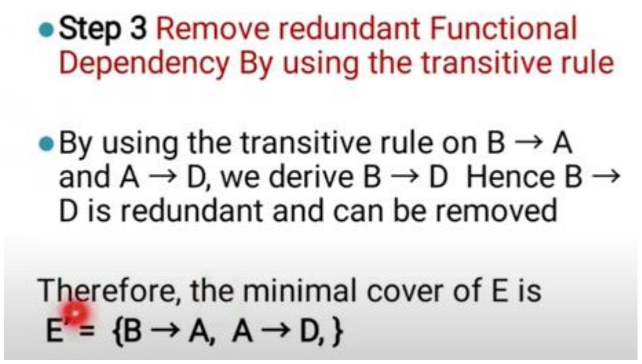


Step 2 : Check if LHS of Functional Dependency has only single attributes

AB → D has two attributes on LHS. Check whether it can be replaced by B→D or A→D.

Thus A→D is redundant will not consider so AB → D may be replaced by B → D.

E' : {B → A, A → D, B → D}



- **Step 3** Remove redundant Functional Dependency By using the transitive rule

- By using the transitive rule on B → A and A → D, we derive B → D Hence B → D is redundant and can be removed

Therefore, the minimal cover of E is
E' = {B → A, A → D, }