CHAPTER - 5

DATABASE CONSTRAINTS AND RELATIONAL DATABASE DESIGN

5.1 Introduction

A database constraint is a rule that governs the data that can be stored in a database. Constraints are used to ensure the accuracy, consistency, and integrity of the data in the database. Constraints are typically defined at the time the database schema is created and can be enforced by the database management system.

5.2 Integrity Constraints

Integrity constraints are the set of rules which are enforced to maintain the quality of information. Constraints are used to limit the type, length and range of data that can go into the table. Accuracy and consistency in the database can be maintained with the help of constraints. If there is any violation between the constraint and the data action, the action is aborted. Thus, integrity constraints help to guard against accidental damage to the database.

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is altered with the ALTER TABLE statement.

a. Constraints on a Single Relation

The create table command may also include integrity-constraint statements. In addition to the primary-key constraint, there are a number of other ones that can be included in the create table command. The allowed integrity constraints include

i. not null
ii. unique
iii. check(<predicate>)
iv. Primary Key Constraint

i. Not Null Constraint

The null value is a member of all domains, and as a result it is a legal value for every attribute in SQL by default. For certain attributes, however, null values may be inappropriate. The not null constraint prohibits the insertion of a null value for the attribute. Consider a tuple in the student relation where name is null. Such a tuple gives student information for an unknown student; thus, it does not contain useful information. Similarly, we would not want the department budget to be null. In cases such as this, we wish to forbid null values, and we can do so by restricting the domain of the attributes name and budget to exclude null values, by declaring it as follows:

name varchar(20) NOT NULL

budget numeric(12,2) NOT NULL

Example:

Employee (EmpId, EmpName, Salary)

EmpId	EmpName	Salary
123	Sudip	50000
142	Kundan	60000
164	Mithun	20000
NULL	Vinod	27000

NULL is not allowed in EmpId as Primary Key can't contain a NULL value.

ii. Unique Constraint

SQL also supports an integrity constraint:

UNIQUE ($A_{j1}, A_{j2}, ..., A_{jm}$)

The unique specification says that attributes $A_{j1}, A_{j2}, ..., A_{jm}$ form a superkey; that is, no two tuples in the relation can be equal on all the listed attributes. However, attributes declared as unique are permitted to be null unless they have explicitly been declared to be not null.

iii. The Check Clause

When applied to a relation declaration, the clause check(P) specifies a predicate P that must be satisfied by every tuple in a relation.

A common use of the check clause is to ensure that attribute values satisfy specified conditions, in effect creating a powerful type system. For instance, a clause check (budget > 0) in the create table command for relation department would ensure that the value of budget is nonnegative.

As another example, consider the following:

```
CREATE TABLE section
(course_id VARCHAR (8),
Sec_id VARCHAR (8),
semester VARCHAR (6),
year NUMERIC (4,0),
building VARCHAR (15),
room number VARCHAR (7),
time_slot_id VARCHAR (4),
PRIMARY KEY (course id, sec id, semester, year),
CHECK (semester IN ('Fall', 'Winter', 'Spring', 'Summer')));
```

Here, we use the check clause to simulate an enumerated type by specifying that semester must be one of 'Fall', 'Winter', 'Spring', or 'Summer'. Thus, the check clause permits attribute domains to be restricted in powerful ways that most programming language type systems do not permit.

iv. Primary Key Constraint

Keys are the attributes that are used to identify an entity within its entity set uniquely. An entity set can have multiple keys, but one key will be primary key. A primary key can contain a unique value in the relational table.

Example:

ID	Name	Semester	Age
1000	Prabesh	I	17
1001	Sudhir	II	24
1002	Akhil	V	21
1003	Athul	III	19
1002	Jay	VIII	22

Not allowed because all row must be unique

To define a PRIMARY KEY constraint on single column

```
CREATE TABLE Persons (
    ID INT NOT NULL PRIMARY KEY,        } Here, defining 'Not Null' is optional
    LastNameVARCHAR(50) NOT NULL,
    FirstName VARCHAR(50),
    Age INT);
```

To define a PRIMARY KEY constraint on multiple columns with constraint name

```
CREATE TABLE Persons (
    ID INT NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    FirstName VARCHAR(50),
    Age INT,
    CONSTRAINT PK_Person PRIMARY KEY (ID, LastName));
```

Here 'PK_Person' is a name of primary key constraint.

5.3 Referential Integrity

Often, we wish to ensure that a value that appears in one relation (the referencing relation) for a given set of attributes also appears for a certain set of attributes in another relation (the

referenced relation). Such conditions are called referential integrity constraints, and foreign keys are a form of a referential integrity constraint where the referenced attributes form a primary key of the referenced relation. A referential integrity constraint is specified between two tables. If a foreign key in Table1 refers to the primary key of Table2, then every value of the foreign key in Table2 must be NULL or available in Table1.

Table 1

Primary Key

D_No	D_Location
11	Muktinath
24	Itahari
13	Budhabare

Table 2

Foreign Key

Emp_ID	Name	Age	D_No
1	Pravin	20	11
2	Suman	40	24
3	Nabin	27	18
4	Balkrishna	38	13

Not allowed

18 is not allowed because it's not present in Table 1.

Example:

1. **To define foreign key on single column**

```
CREATE TABLE Orders (
OrderID INT PRIMARY KEY,
OrderNumber INT NOT NULL,
PersonID INT FOREIGN KEY REFERENCES Persons (PersonID));
```

2. **To define foreign key on multiple column with constraint name**

```
CREATE TABLE Orders (
OrderID INT PRIMARY KEY,
```

OrderNumber INT NOT NULL,

PersonID INT,

LastName VARCHAR(50),

CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID, LastName)

REFERENCES Persons(PersonID, LastName));

Here 'FK_PersonOrder' is a name of foreign key constraint.

Cascading in Referential Integrity

When a referenced record in a parent table is modified or removed, cascading, a characteristic of referential integrity in database systems, decides what should happen. When cascade is enabled for a foreign key constraint, changes made to the parent table are automatically propagated to any associated child tables.

There are two common types of cascading actions in referential integrity:

i. . **Cascading Updates**

Cascading updates automatically propagate changes to the matching foreign key values in the child table(s) when a referenced record in the parent database is updated. By doing this, it is made sure that the relationships' integrity is upheld.

For example, if we update the primary key value of a customer record in the parent table, the cascading update would automatically update the foreign key values in the orders table that reference that customer.

ii. **Cascading Deletes**

Cascading deletes automatically remove the associated child records in the child table(s) when a referenced record in the parent table is deleted. By deleting any children records that rely on the parent record that was deleted, this guarantees that the database will stay consistent.

For example, if you delete a department record from the parent table, the cascading delete would automatically remove all the associated employee records from the child table.

It's important to remember that enabling cascading actions should be done cautiously because they can significantly affect the database's consistency and integrity. If cascading updates or deletes are to be enabled, careful study, understanding of the data linkages, and business requirements are required.

By using cascading actions in referential integrity, database administrators can automate the maintenance of data relationships and ensure that changes to the parent table are properly reflected in the child tables, simplifying the management of the database and reducing the risk of inconsistent data.

5.4 Assertions and Triggers

5.4.1 Assertion

An assertion is a predicate expressing a condition that we wish the database always to satisfy. Domain constraints and referential integrity constraints are special forms of assertions. There are many constraints that we cannot express by using only these special forms.

Two example of such constraints are:

i. The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch

ii. Every loan has at least one customer who maintains an account with a minimum balance of 100000

Syntax:

Create assertion <assertion_name> check <predicate>;

Predicate must return true or false.

Example:

Creating a table

CREATE TABLE Table_1 (column_1 INT, column_2 VARCHAR(4));

Creating an assertion

CREATE ASSERTION constraint_1

CHECK ((SELECT AVG(column_1) FROM Table_1) > 50)

Suppose column_1 has single row with value 52.

Inserting Value in column_1:

```
INSERT INTO Table_1(column_1)
VALUES (46);
```

This statement will violate the constraint_1 because the condition inside check clause returns false value i.e. 46>50 is false. So, the value wouldn't get inserted.

When an assertion is created, the system tests it for validity. If the assertion is valid, any further modification to the database is allowed only if it does not cause that assertion to be violated. This testing may introduce a significant amount of overhead; hence assertions should be used with great care.

5.4.2 Trigger

A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database.

To design a trigger mechanism, we must meet two requirements :

i. Specify the conditions under which the trigger is to be executed. This is broken up into an event that causes the trigger to be checked and a condition that must be satisfied for trigger execution to proceed.

ii. Specify the actions to be taken when the trigger executes

There are basically three parts of a trigger:

i. **Event**
When this event happens, the trigger is activated.

ii. **Condition (optional)**
If the condition is true, the trigger executes, otherwise it is skipped.

iii. **Action**
The actions performed by the trigger

Syntax of Trigger:

```
CREATE TRIGGER
trigger_name
trigger_time
trigger_event
ON
table_name FOR EACH ROW
BEGIN
...
END;
```

trigger_time------> BEFORE, AFTER

trigger_event------> INSERT, UPDATE, DELETE

Example:

1. **BEFORE INSERT Trigger**

#First of all we create a table named customers and then we defined trigger named age_verify to that table.

```
CREATE TABLE customers(cust_id INT, age INT, name VARCHAR(30));

#define trigger
DELIMITER //
CREATE TRIGGER age_verify
BEFORE INSERT ON customers
FOR EACH ROW
IF new.age < 0 THEN SET new.age =0;
END IF; //

#Now let's insert the values to the customers
INSERT INTO customers VALUES (101, 27, "James"),
(102, -40, "Ammy"),
(103, -32, "Ben"),
(103, -39, "Angela");
```

In the customers table, for those customers: 102 and 103, the trigger automatically set the age to 0 since their age is less than 0. This is how before insert trigger works.

2. **AFTER INSERT Trigger**

#First of all we create a table named customers1 and message then we defined trigger named check_null_dob to that table.

```
CREATE TABLE customers1(
id INT AUTO_INCREMENT PRIMARY KEY,
name VARCHAR(40) NOT NULL,
email VARCHAR (30),
birthdate DATE);

CREATE TABLE message (
id INT AUTO_INCREMENT ,
messageId INT,
message VARCHAR(300) NOT NULL,
PRIMARY KEY (id,messageId));

DELIMITER //
CREATE TRIGGER check_null_dob
AFTER INSERT
ON customers1
FOR EACH ROW
BEGIN
IF new.birthdate IS NULL THEN
INSERT INTO message(messageId, message)
VALUES(new.id, concat('Hi', new.name, ',Please update your date of birth'));
END IF;
END//

INSERT INTO
customers1 (name, email, birthdate)
VALUES
('John Doe', 'john.doe@example.com ', NULL),
('Jane Doe', 'jane.doe@example.com', '2000-01-01');
SELECT *
FROM customers1 ;
```

id	Name	Email	birthdate
1	John Doe	john.doe@example.com	(NULL)
2	Jane Doe	jane.doe@example.com	2000-01-01

```
SELECT *
FROM message;
```

Id	Messageid	Message
1	1	Hi John Doe please update your Date of Birth

3. **BEFORE UPDATE Trigger**

```
#create employees table
CREATE TABLE employees(emp_id INT AUTO_INCREMENT
PRIMARY KEY,
emp_name VARCHAR(25),
age INT,
salary FLOAT);

#insert some values to employees table
INSERT INTO employees VALUE
(),
();
#above lines insert default values
#create trigger named: upd_trigger
DELIMITER //
CREATE TRIGGER upd_trigger
BEFORE UPDATE
ON employees
FOR EACH ROW
```

```
BEGIN
SET new.salary=CASE
WHEN new.salary >10000 THEN 8500
WHEN new.salary < 10000 THEN 7200
END;
END
//

#Now perform update command
UPDATE employees
SET salary = 80000;

#we can see that new salary is 80000 which is greater than
10000. the trigger updates the salary as 85000
SELECT *
FROM employees;
```

emp_id	emp_name	Age	salary
1	NULL	NULL	85000
2	NULL	NULL	85000

4. BEFORE DELETE Trigger

```
#create a table named salarydel to store deleted details
CREATE TABLE salarydel(emp_id INT ,
emp_name VARCHAR(25),
age INT,
salary FLOAT);
#create trigger named: salary_delete
DELIMITER //
CREATE TRIGGER salary_delete
BEFORE DELETE
ON employees
FOR EACH ROW
```



```
BEGIN
INSERT INTO salarydel(emp_id , emp_name ,age,salary)
VALUES( old.emp_id , old.emp_name , old.age,old.salary);
END //

#Now delete emp_id =1 from employees
DELETE FROM employees
WHERE eid =1;

#successfully deleted
SELECT *
FROM employees;
```

emp_id	emp_name	age	salary
2	NULL	NULL	85000

```
SELECT *
FROM salarydel;
```

emp_id	emp_name	age	salary
1	NULL	NULL	85000

5.5 Functional Dependencies

Functional dependency is a relationship that exists when one attribute uniquely determines another attribute. If R is a relation with attributes X and Y, a functional dependency between the attributes is represented as X→Y, which specifies Y is functionally dependent on X. Here X is a determinant set and Y is dependent attribute. Each value of X is associated with precisely on Y value. Functional dependency in a database serves as a constraint between two set of attributes.

Defining functional dependency is an important part of relational database design and contributes to aspect normalization. Functional dependency typically exists between the primary key and non-key attribute within a table

$X \rightarrow Y$

The left side of FD is known as determinant, the right side of the production is known as dependent

Example:

Assume we have an employee table with attributes: Emp_Id, Emp_Name, Emp_Address

i.e. employee(Emp_Id,Emp_Name,Emp_Address)

Emp_ID	Emp_Name	Emp_Address
1234	Bishon	Parbat
4567	Nabin	Butwal

Here Emp_Id attribute can uniuely identify the Emp_Name attribute of employee table because if we know the Emp_Id, we can tell the employee's name associated with it.

Functional dependency can be written as:

Emp_Id→ Emp_Name

We can say that Emp_Name is functionally dependent on Emp_Id

Compound Determinants

If more than one attributes are necessary to determine other attribute in an entity, then such determinant is termed as compound or composite determinant.

Full Functional Dependency

This is the situation when it is necessary to use all the attributes of composite determinant to identify its object uniquely. A and B are attributes of a relation, B is fully functionally dependent on A if B is functionally dependent on A but not on any proper subset of A.

Partial Functional Dependency

This is the situation that exist if it is necessary to only use a subset of the attributes of the composite determinant to identify its object uniquely. A functional dependency A→B is partially dependent if there are some attributes that can be removed from A and yet the dependency holds.

Types of Functional Dependency:

i. Trival functional dependency
ii. Non-trival functional dependency

i. **Trival Functional Dependency**

A→B has trivial dependency if B is a subset of A. The following dependencies are also trivial:

A→A, B→B

Example:

Consider a table with two columns Employee_Id and Employee_Name

{Employee_Id, Employee_Name}→ Employee_Id is a trivial functional dependency as:

Employee_Id is subset of {Employee_Id, Employee_name}

Also,

Employee_Id→ Employee_Id

Employee_Name → Employee_Name are trivial dependencies too.

ii. **Non-trivial Functional Dependency**

A → B has non trivial functional dependency if B is not a subset of A.

When A intersection B is NULL, then A→B is called as complete non-trivial.

Company	CEO	Age
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Apple	Tim Cook	58
Amazon	Andy Jassy	55

Example:

ID → Name

Name → DOB

Company → CEO

But CEO is not a subset of company and hence non-trivial functional dependency.

Armstrong's Axioms property of functional dependency

Armstrong's Axioms property was developed by William Armstrong in 1974 to reason out about functional dependencies.

i. **Transitivity**

if A→B and B→C, then A→C i.e transitive relation.

ii. **Reflexibity**

A→B, if B is a subset of A.

iii. **Augmentation**

The last rule suggests: AC→BC, if A→B.

Characteristics of Functional Dependency

There is one-to-one relationship between the attributes on the left-hand side(determinant) and those on the right-hand side of a functional dependency and this relation holds for all time. The determinants have the minimal number of attributes necessary to maintain the dependency with the attributes on right hand side.

Closure of a set of Functional Dependencies

A closure is a set of all possible FDs that can be derived from given set of FDs. Closure set is also referred as a complete set of FDs. If F is used to denote the set of FDs for relation R then a closure of a set of FDs implied by F is denoted by F+. Let's consider the set F of functional dependencies given below:

$F = \{A \rightarrow B, B \rightarrow C, C \rightarrow D\}$

From F it is possible to derive the following dependencies:

A→A

A→B

A→ C

A→D

A→ ABCD

A+→ ABCD (Union)

(Closure of A under F)

All such type of FDs derived from each FD of F form closure of F

Additional Rules

i. **Union Rule**

if $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds then $\alpha \rightarrow \beta\gamma$ holds

ii. **Decomposition Rule**

if $\alpha \rightarrow \beta\gamma$ holds then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ holds

iii. **Pseudotransitivity Rule**

if $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds then $\alpha\ \gamma \rightarrow \delta$ holds

Steps to determine F+

- Determine each set of attributes X that appears as a left hand side of some FD in F.
- Determine the set X+ of all attributes that are dependent on X.
- All such set of X+, in combine form closure of F.

Example:

For given FD, find all possible F+

R= {A, B, C, G, H, I)

F= {A→B, A→C, CG→H, G→I, B→H, CG→I}

Some members of F+

A→H from A→B and B→H by transitivity

AG→I by augmenting A→C with G to get AG→CG and then transitivity with CG→I

CG→HI by augmentation

Transitive Dependency

A functional dependency is said to be transitive if it is indirectly formed by two functional dependencies.

Example:

X→Z is a transitive dependency if the following three functional dependencies hold true:

X→Y

Y does not →X

Y→Z

A transitive dependency can only occur in a relation of three of more attributes.

Book	Author	Author_Age
Game Of Thrones	George R. R. Martin	72
Harry Potter	J. K. Rowling	55
Dying Of The Light	George R. R. Martin	72

{Book} → {Author} (if we know the book, we know the author name)

{Author} does not → {Book}

{Author} → {Author_age}

Therefore as per the rule of transitive dependency:

{Book} → {Author_age} should hold, that makes sense because if we know the book name we can know the author's age.

5.6 Multivalued Dependencies

	A	B	C
	Name	**Project**	**Hubby**
T1	Yuvaraj	MS	Reading
T2	Yuvaraj	Oracle	Music
T3	Yuvaraj	MS	Music
T4	Yuvaraj	Oracle	Reading

Here project and hubby are multivalued attributes because they contain the different values for the same name (Yuvaraj)

Attributes (Columns): A, B, C

Tuples (Rows): T1, T2, T3, T4

R = set of attributes, r = relation

We can say that multivalued dependency exists if the following conditions are met.

Any attribute say 'A' multiple define another attribute B; if any legal relation r(R), for all pairs of tuples T1 and T2 in r, such that,

T1[A] = T2[A]

Then there exists t3 and t4 in r such that.

T1[A] = T2[A] = T3[A] = T4[A]

T1[B] = T3[B]

T3[R–B] = T2 [R –B]

T2[B] = T4[B]

T4[R–B] = T1 [R–B]

Then multivalued (MVD) dependency exists.

To check the MVD in given table, we apply the conditions stated above and we check it with the values in the given table.

T1[A] = T2[A] = T3[A] = T4[A] =Yuvaraj

T1[B] = T3[B]=Ms

T3[R–B] = T2 [R–B]=Yuvaraj, Music

T2[B] = T4[B]=Oracle

T4[R–B] = T1 [R–B]=Yuvaraj, Reading

Hence, we know that MVD exists in the above table and it can be stated by,

name →→ project

name →→ hobby

5.7 Normalization and Normal Forms

Normalization is the process of restructuring the logical data model of a database to eliminate redundancy, organize data efficiently and reduce repeating data and to reduce the potential for anomalies during data operations.

Database Normalization is a technique of organizing the data in the database. Normalization generally involves splitting existing tables into multiple ones which must be rejoined or link each time a query is issued. Normalization is multi-step process. Normalization is the process of efficiently organizing data in database with two goal:

i. Eliminate redundant data

ii. Ensure data dependencies make sense

Data normalization also may improve data consistency and simplify future extension of the logical data model. The formal classifications used for describing a relational database's level of normalization are called normal forms Normal Forms is abbreviated as NF.

What happens without Normalization?

- A non-normalized database can suffer from data anomalies.
- A non-normalized database may store data representing a particular referent in multiple locations.
- An update to such data in some but not all of those locations results in an update anomaly, yielding inconsistent data.
- A non-normalized database may have inappropriate dependencies, i.e. relationships between data with no functional dependencies.

Example:

Roll_No	Name	Branch	HOD	Office_Tel
1	Jagdish	BCT	Hari	01-5337
2	Yagya	BCT	Hari	01-5337
3	Suman	BCT	Hari	01-5337
4	Shankar	BCT	Hari	01-5337

Insertion Anomaly

Suppose for a new admission until and unless a student opts for a branch, data of the student cannot be inserted or else we will have to set the branch information as NULL.

If we have to insert data of 100 student of same branch then the branch information will be repeated for all those 100 students.

Updation Anomaly

If Hari is no longer the HOD of computer department or what if hari leaves the college?

In that case all student records will have to be updated and if by mistake we miss any record it will lead to data inconsistency.

Deletion Anomaly

In our student table two different information are kept together, student information and branch information.

Hence at the end of the academic year if student records are deleted we will also lose the branch information. This is deletion anomaly.

Advantages of Normalization

- Elimination of redundant data storage.
- Closed modeling of real world entities, processes and their relationship.
- Structuring of data so that model is flexible.
- Less storage space.
- Easier to add data and avoid certain updating anomalies.
- Facilitate the enforcement of data constraints.

Normalized Database

Normalized databases have a design that reflects the true dependencies between tracked quantities, allowing quick updates to data with little risk of introducing inconsistencies. Instead of attempting to lump all information into one table, data is spread out logically into many tables.

Normalizing the data is decomposing a single relation into a set of smaller relations which satisfy the constraints of the original relation. Redundancy can be solved by decomposing the tables. However certain new problems are caused by decomposition. Normalization helps us to make a conscious decision to avoid redundancy keeping the pros and cons in mind.

One can only describe a database as having a normal form if the relationships between quantities have been rigorously defined. It is possible to use set theory to express this knowledge once a problem domain has been fully understood, but most database designers model the relationships in terms of an "idealized schema". (The mathematical support came back into play in proofs regarding the process of transforming from one

form to another.) The transformation of conceptual model to computer representation format is known as normalization.

Stages of Normalization

First Normal Form(1NF)

A relational schema R is in first Normal form if the domains of all attributes of R are atomic. Non-atomic values complicate storage and encourage redundant(repeated) storage of data. To transform the un normalized table(a table that contains one or more repeating groups) to first normal form, identify and remove the repeating groups within the table, (i.e multi valued attributes, composite attributes and their combinations).

Un Normalized Table

Emp_id	Ename	Project
1	Subodh	A, B
2	Anuj	C, D

Table in 1 NF

Emp_id	Ename	Project
1	Subodh	A
2	Subodh	B
3	Anuj	C
4	Anuj	D

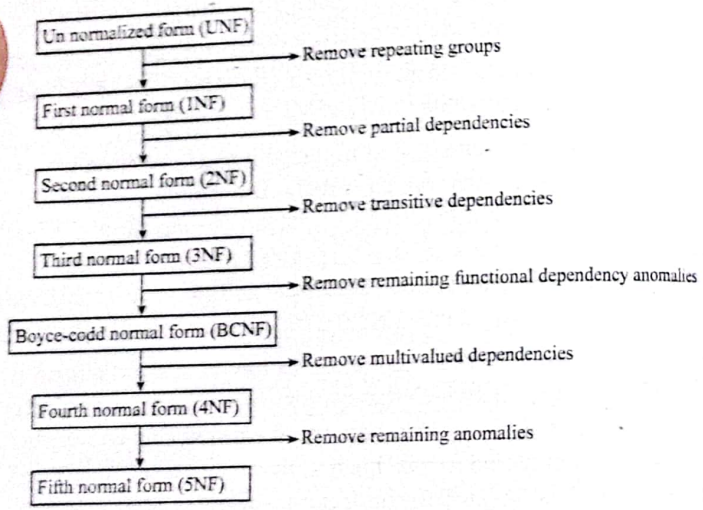

Figure: Stages of normalization

Second Normal Form (2NF)

A table is in 2NF if and only if it is in 1NF and has no attributes which require only part of the key to uniquely identify them. For a table to be in 2NF we need to remove partial dependency. Where a key has more than one attribute, check that each non key attribute depends on the whole key and not part of the key

For each subset of the key which determines an attribute or group of attributes create new form. Move the dependant attributes to the new form. Add the part key to new form, making it the primary keyMark the part key as a foreign key in the original form.

Example:

Department(Project_Name, Employee_Name, Emp_Hire_Date, Project_Manager)

Project_Name and Employee_Name are the candidate key for this table. Emp_Hire_Date and Project_Manager are partially depend on the Employee_Name, but not depend on the Project_Name. Therefore, this table will not satisfy the 2NF

In Order to satisfy the second Normal form, We need to put the Emp_Hire_Date and Project Manager to other tables. We can put the Emp_Hire_Date to the Employee table and put the Project_Manager to the Project table.

So now we have three tables:

Department(Project_Name, Employee_Name)

Project(Project_ID, Project_Name, Project_Manager)

Employee(Employee_ID, Employee_Name, Employee_Hire_Date)

Now, the Department table will only have the candidate key left.

Emp_id	Proj_id	Ename	Pname
1	Erp_100	Subodh	HRMS
1	Erp_200	Subodh	ETRMS
2	Erp_100	Anuj	HRMS
2	Erp_200	Anuj	ETRMS

Here, Pname can be uniquely determined by proj_id only so pname is partially dependent on proj_id.

Converting into 2nd normal form | Removing partial dependency:

For table to be in 2nd normal form we need to remove partial dependency.

Emp_id	Proj_id	Ename
1	Erp_100	Subodh
1	Erp_200	Subodh
2	Erp_100	Anuj
2	Erp_200	Anuj

Proj_id	Pname
Erp_100	HRMIS
Erp_200	ETRMS

Third Normal Form (3NF)

A relation is in third normal form if and only if it is :

- In second normal form
- Every non key attribute is non transitively dependent on the primary key.

An attribute C is transitively dependent on attribute A if there exists an attribute B such that A→B and B→C, then A→C

Emp_id	Ename	Position	Dname	Dhead
1	Subodh	Software Engineer	IT	Bidur
2	Rajesh	System Admin	SA	Anup

Dname and Dhead are separated and a new table is made and new column deptno is added to it.

Emp_id	Ename	Position	Deptno
1	Subodh	Software Engineer	1
2	Rajesh	System Admin	2

Deptno	Dname	Dhead
1	IT	Bidur
2	SA	Anup

Boyce Codd Normal Form (BCNF)

A relation is in BCNF if for every FD A→B either

- B is contained in A(the FD is trivial), or
- A contains a candidate key of the relation

In other words:

Every determinant in a non-trivial dependency is a key.

If there is only one candidate key then 3NF and BCNF are the same.

Emp_id	Project	phead
1	HRMIS	Manju
1	ETRMS	Bidur
2	CTRMS	Rajeev
2	FARRMS	Deva

Here, emp_id and project together makes a primary key. This table satisfies 1st, 2nd and 3rd normal form but it doesn't satisfies BCNF.

Why the table doesn't satisfy BCNF?

To satisfy BCNF, table must be in 3rd normal form and for all FD: X → Y within the table X must be super key (prime attribute). It satisfies 1st condition but not the second one.

Since, emp_id and project together makes a primary key emp_id and project both are prime attributes.

In the table, phead attribute determines project since each project head leads only one project at a time. ie (1-1 relationship). As project head is a non prime attribute it fails the 2nd condition.

Emp_id	Phead_id
1	1
1	2
2	3
2	4

Phead_id	Phead	Project
1	Manju	HRMIS
2	Bidur	ETRMS
3	Rajeev	CTRMS
4	Deva	FARRMS

Fourth Normal Form (4NF)

For a table to satisfy fourth normal form it should satisfy following two conditions:

i. It should be in BCNF

ii. And the table should not have any multi-valued Dependency

S_Id	Course	Hobby
1	Science	Cricket
	Maths	Hocky
2	C#	Cricket
	Php	Hockey

S_Id 1 has opted for two courses, science and Maths and has two hobbies cricket and hockey which will lead to the following table.

S_Id	Course	Hobby
1	Science	Cricket
1	Maths	Hockey
1	Science	Hockey
1	Maths	Cricket

Here course and hobby are independent of each other

CourseOpted Table

S_Id	Course
1	Science
1	Maths
2	C#
2	Php

Hobbies Table

S_Id	Hobby
1	Cricket
1	Hockey
2	Cricket
2	Hockey

5.8 Decomposition of Relation Schemas

When a relation in the relational model is not in appropriate Normal Form then the decomposition of relation is required. If the relation has no proper decomposition then it may lead to problems like loss of information. Decomposition is used to eliminate some of the problems of bad design anomalies, inconsistencies and redundancy

Types of Decomposition

1. Lossless Decomposition
2. Lossy Decomposition

1. Lossless Decomposition

Let r (R) be a relation schema, and let F be a set of functional dependencies on r (R). Let R 1 and R 2 form a decomposition of R. We say that the decomposition is a lossless decomposition if there is no loss of information by replacing r (R) with two relation schemas r 1 (R 1) and r 2 (R 2)

The decomposition is lossless if, for all legal database instances (that is, database instances that satisfy the specified functional dependencies and other constraints), relation r contains the same set of tuples as the result of the following SQL query:

```
SELECT *
FROM (SELECT R 1 FROM r)
NATURAL JOIN (SELECT R 2 FROM r);
```

This is stated more succinctly in the relational algebra as:

$\Pi_{R1}(r) \bowtie \Pi_{R2}(r) = r$

In other words, if we project r onto R 1 and R 2 , and compute the natural join of the projection results, we get back exactly r.

To check for lossless join decomposition using FD set, following conditions must hold:

1. Union of Attributes of R1 and R2 must be equal to attribute of R. Each attribute of R must be either in R1 or in R2.

   Att(R1) U Att(R2) = Att(R)

2. Intersection of Attributes of R1 and R2 must not be NULL.

   $Att(R1) \cap Att(R2) \neq \phi$

3. Common attribute must be a key for at least one relation (R1 or R2)

$Att(R1) \cap Att(R2) \rightarrow Att(R1)$ or $Att(R1) \cap Att(R2) \rightarrow Att(R2)$

Example:

A relation R (A, B, C, D) with FD set{A→BC} is decomposed into R1(ABC) and R2(AD) which is a lossless join decomposition as:

First condition holds true as:

$Att(R1) \cup Att(R2) = (ABC) \cup (AD) = (ABCD) = Att(R)$.

Second condition holds true as:

$Att(R1) \cap Att(R2) = (ABC) \cap (AD) \neq \phi$

Third condition holds true as:

$Att(R1) \cap Att(R2) = A$ is a key of R1(ABC) because A→BC is given.

Decomposition of R = (A, B, C)

R1 = (A, B) R2 = (B, C)

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

$\Pi_{A,B}(r)$

B	C
1	A
2	B

$\Pi_{B,C}(r)$

$\Pi_A(r) \bowtie \Pi_B(r)$

A	B	C
α	1	A
β	2	B

2. Lossy Decomposition

As the name suggests, when a relation is decomposed into two or more relational schemas, the loss of information is unavoidable when the original relation is retrieved.

Consider there is a relation R which is decomposed into sub relations $R_1, R_2, \ldots, R_n$.

This decomposition is called lossy join decomposition when the join of the sub relations does not result in the same relation R that was decomposed.

The natural join of the sub relations is always found to have some extraneous tuples.

For lossy join decomp

$R1 \bowtie R2 \bowtie R3 \ldots\ldots \bowtie Rn \supset R$

As an example of a lossy decomposition, the decomposition of the employee schema into employee1 and employee2

employee (ID, name, street, city, salary)

employee1 (ID, name)

employee2 (name, street, city, salary)

The result of employee1 ⋈ employee2 is a superset of the original relation employee, but the decomposition is lossy since the join result has lost information about which employee identifiers correspond to which addresses and salaries, in the case where two or more employees have the same name.

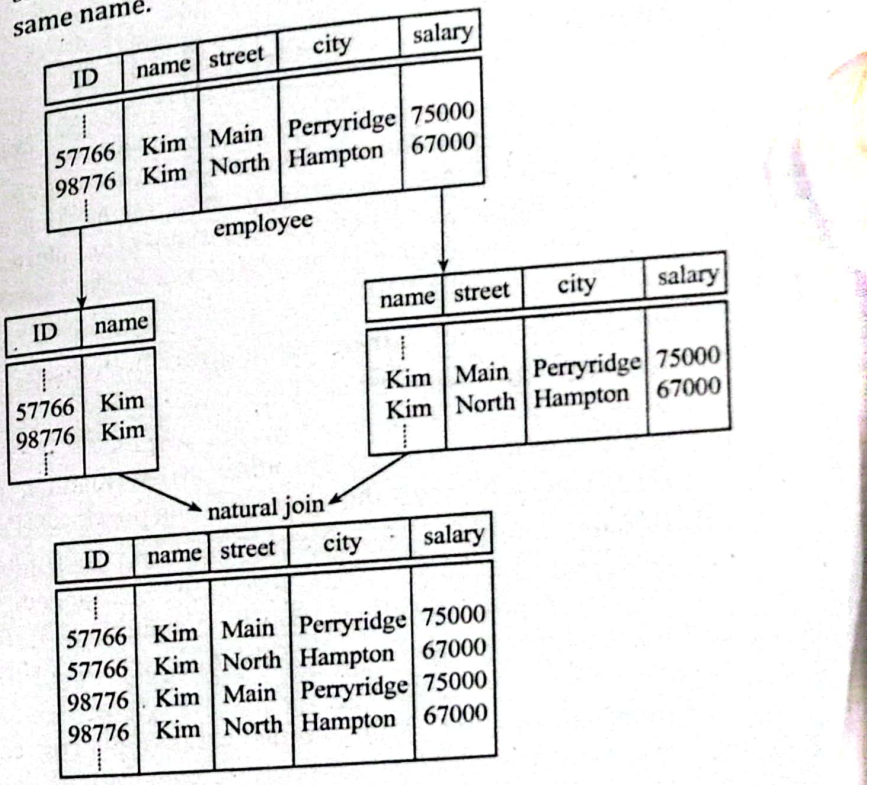

Figure: Lossy decomposition

SOLUTION TO EXAMS' AND OTHER IMPORTANT QUESTIONS

1. **Consider the following relation where {M_ID and P_ID} are primary keys. State in which normal form the relation is. What anomalies can occur in this relation? How can these anomalies be removed?**

M_ID	M_Date	P_ID	Quantity
M11	16 June, 2022	11	20
M11	26 June, 2022	16	30
M22	3 September, 2022	15	20
M22	13 September, 2022	16	60
M22	23 September, 2022	12	35

[2022 Fall]

ANS: As each attribute has atomic values and there are no repeated groups, this relation is currently in the First Normal Form (1NF). Due to partial dependencies, it does not, however, satisfy the Second Normal Form (2NF). i.e., M_Date does not fully depend on {M_ID,P_ID}.

Anomalies that can occur in this relation include:

i. **Insertion Anomaly:** If we don't have all the necessary data, we can't insert a new row. For instance, we cannot add a row if we wish to introduce a new manufacturer (M_ID) but don't know the corresponding purchase (P_ID).

ii. **Update Anomaly:** Multiple rows must be updated whenever the value of an attribute is changed. For instance, updating all the rows containing a P_ID is required if we wish to change the quantity for a certain product (P_ID).

iii. **Deletion Anomaly:** Unintentionally deleting a row could result in the loss of data that is required for other purposes. For instance, if we remove the purchase data

entry with P_ID = 15, the data regarding the manufacturer made on September 3, 2022 (M_ID = M22) is lost.

To remove these anomalies, we can normalize the relation further. Here's one possible approach:

Create a separate table for "Manufacture_Details" with attributes: M_ID and M_Date and {M_ID, M_Date} as primary key.

Manufacture_Details Table:

M_ID	M_Date
M11	16 June, 2022
M11	26 June, 2022
M22	3 September, 2022
M22	13 September, 2022
M22	23 September, 2022

Create another table for "Purchase_Details" with attributes: M_ID (foreign key referencing Manufacture_Details table), P_ID (foreign key referencing the relevant products table), and Quantity. Primary key would be {M_ID,P_ID}.

Purchase_Details Table:

M_ID	P_ID	Quantity
M11	11	20
M11	16	30
M22	15	20
M22	16	60
M22	12	35

By separating the data into two tables, we eliminate the partial dependencies and achieve Second Normal Form (2NF). This structure allows us to insert, update, and delete data without encountering the aforementioned anomalies.

2. When do you use triggers? [2022 Fall]

ANS: Triggers in the database are used to automatically execute a set of actions or instructions in response to certain events or changes in the database. Triggers are typically used in the following scenarios:

i. Enforcing Data Integrity

Triggers can be used to enforce complex business rules and data validation constraints that cannot be easily expressed using standard constraints and rules. For example, we can create a trigger that checks that certain conditions are met before allowing an insert or update operation.

ii. Auditing and Logging

For auditing purposes, triggers can be used to record changes made to particular tables or columns. They can automatically log information about modifications, including who made them, when they happened, and what was modified.

iii. Synchronization of Data

Data synchronization between various tables or databases is possible with the help of triggers. To preserve data consistency, for instance, a trigger can be used to automatically update related tables when a change is made in one table.

iv. Business Logic Enforcement

Specific business logic or rules that are difficult to define through constraints can be enforced with the help of triggers. This could include complex calculations, derived values, or workflow-related operations.

v. Data Transformation

Prior to being entered, modified, or deleted, data can be transformed using triggers. This can involve applying particular formatting standards, normalizing the data, or converting the data.

3. What is the role of Triggers? Write an SQL trigger to carry out the following action: On delete of an account, for each owner of the account, check if the owner has any remaining amount, and if she does not, delete her from the *depositor* relation. [2021 Fall]

ANS: The role of a trigger is to enforce certain business rules, maintain data integrity, or perform other automated actions based on changes in the database.

```
DELIMITER //
CREATE TRIGGER delete_owner_trigger
AFTER DELETE ON account
FOR EACH ROW
  BEGIN
  -- Declare a variable to hold the count of remaining
  accounts
  -- for the owner
  DECLARE remaining_accounts INT;

  -- Check if the owner has any remaining amount
  SELECT COUNT(*) INTO remaining_accounts FROM
  depositer WHERE account_number =
  OLD.account_number and old.balance=0;

  -- If the owner has no remaining accounts, delete her from
  --the depositor relation
  IF remaining_accounts = 1 THEN
        DELETE FROM depositer WHERE account_number
        =OLD.account_number;
  END IF;
  END
//DELIMITER ;
```

4. Illustrate redundancy and the problems that it can cause. [2020 Spring]

ANS: Redundancy refers to the duplication or repetition of data within a database or system. It occurs when the same information is stored multiple times in different locations or

in multiple records within the same location. While a certain level of redundancy is sometimes necessary for efficient data retrieval and system performance, excessive redundancy can lead to various problems and challenges. Here are some problems that redundancy can cause:

a. Data Inconsistency

The chance of data errors or inconsistencies rises with redundancy. Any update, insertion, or deletion of data that is stored in several locations must be carefully handled to keep all copies of the data synchronized. There may be discrepancies and faults in the system if one copy of the data is changed but others are not correctly updated.

b. Increased Storage and Maintenance Costs

Redundancy uses more storage space and raises the price of maintaining and storing data. The same data must be stored more than once, consuming additional resources and storage space. Higher operational costs are caused by the complexity and time required for updating and keeping redundant data.

c. Update Anomalies

Update anomalies—inconsistencies that happen when redundant data is updated—can be introduced by redundancy. For instance, updating the address in one record but not the others can cause inconsistencies and confusion if the same customer address is stored in several records and the address changes.

d. Decreased Data Integrity

Issues with data integrity are more likely when there is redundant data. Enforcing constraints, integrity rules, and referential integrity becomes more difficult when data is copied across several locations. As a result, the system's dependability and accuracy may suffer from inaccurate or inconsistent data.

e. Poor Performance and Efficiency

Excessive redundancy can affect the effectiveness and performance of a system. The complexity of searches and transactions is increased by the additional processing needed to retrieve and update redundant data. This may result in a decrease in system performance and general efficiency.

f. Difficulties in Data Integration

When integrating data from several sources or systems, redundant data might provide problems. When redundant data is maintained independently by each source, combining and resolving the data is difficult and prone to mistakes. A unified and correct representation of the data may be difficult to achieve due to inconsistencies and conflicts.

g. Increased Risk of Data Anomalies

Redundancy can result in data anomalies, when the system stores false or misleading information. Redundant data can contain flaws and inconsistencies that can spread across the system and result in inaccurate analysis, judgments, and reporting.

5. Explain the role of Functional Dependency in the process of Normalization. [2020 Fall]

ANS: Functional dependency helps in identifying and eliminating data redundancy by ensuring that each attribute in a database table depends on the primary key of the table.

In a database table, if an attribute X is functionally dependent on an attribute Y, it means that for every value of Y, there is a unique value of X. In other words, knowing the value of Y, we can determine the value of X. Functional dependency helps in identifying redundant data because if two attributes are functionally dependent on each other, it means that they contain the same information and one of them can be removed without losing any information.

6. Illustrate the advantage of 4NF with suitable example. [2020 Fall]

ANS: Fourth Normal Form (4NF) is a level of database normalization that deals with multi-valued dependencies.

To illustrate the advantage of 4NF, let's consider an example of a database for a library. Suppose we have the following table of information about books:

Books(BookID, Title, Author, Genre, Publisher, DatePublished)

In this table, the primary key is "BookID". There is no partial dependency or transitive dependency, so it satisfies the third normal form. However, it has a multi-valued dependency between the "Title" and "Author" attributes, as a book can have multiple authors and an author can have written multiple books. This means that the table violates the fourth normal form.

To convert this table into 4NF, we can decompose it into two tables:

Books(BookID, Title, Genre, Publisher, DatePublished)
Authors(BookID, Author)

In this case, the "Books" table contains information about books, including the book's title, genre, publisher, and date of publication. The "Authors" table contains information about the authors, including the author's name and the books they have written.

This decomposition eliminates the multi-valued dependency and ensures that the table is in 4NF. This is advantageous because it reduces redundancy and simplifies the process of updating the database. For example, if an author publishes a new book, we only need to add a new row to the "Books" table and a new row to the "Authors" table, rather than updating multiple rows in a single table. This makes it easier to maintain data consistency and avoid data anomalies.

---

7. Differentiate between Functional Dependency and Multi Valued Dependency. [2019 Spring]

ANS: Functional Dependency refers to a relationship between two or more attributes in a database table, where the value of one attribute determines the value of another attribute. More specifically, if attribute B is functionally dependent on

However, there is a transitive dependency between the "Name" and "Salary" attributes, as the "Salary" attribute is dependent on the "Name" attribute through the "Post" attribute. This means that the "Salary" attribute is not directly dependent on the primary key "Name", and it violates the third normal form.

Also, there is a transitive dependency between the "Name" and "Address" attributes, as the "Address" attribute is dependent on the "Name" attribute through the "Phone" attribute. This means that the "Address" attribute is not directly dependent on the primary key "Name", and it violates the third normal form.

To remove these transitive dependencies and convert the relation into 3NF, we can decompose the original relation into three separate relations:

- Relation 1: Name, Phone, Post
- Relation 2: Post, Salary
- Relation 3: Phone, Address

Relation 1 satisfies the third normal form as all the attributes are directly dependent on the primary key "Name". Relation 2 and Relation 3 also satisfy the third normal form as there are no transitive dependencies.

The final relations in 3NF will look like this:

Relation1:

Name	Phone	Post
Gill	456789	Engineer
Van	654321	Engineer
Robert	456789	Engineer
Brown	654321	Overseer
Albert	454545	Officer

Relation 2:

Post	Salary
Engineer	20000
Engineer	20000
Engineer	20000
Overseer	10000
Officer	10000

Relation 3:

Phone	Address
456789	KTM
654321	BKT
454545	KTM

9. Suppose we are given Schema R = {A, B, C, G, H, I} and set of functional dependencies F = {A→B, A→C, CG→H, B→H, CG→I}. Find the closures of functional dependency F. [2019 Fall]

ANS: We can derive the closures of the given functional dependencies as follows:

A → A

A → B

A→H(A→B,B→H)

A → C

CG→H

AG → H (Using Pseudotransitivity Rule i.e., A → C,CG→H then AG → H)

CG→I

AG → I (Using Pseudotransitivity Rule i.e., A → C,CG→I then AG → I)

B → B

B→H

C → C

G → G

H → H

I → I

Therefore, the closures of functional dependency F are:

F+={A → B, B→H, A→H, A → C, CG→H, AG → H, CG→I, AG → I}

10. Compare and Contrast assertion & triggers. [2018 Spring]

ANS: Assertions and triggers are two important database concepts that are used for ensuring data integrity and enforcing business rules. Though they serve similar purposes, there are some differences between them.

Assertions are declarative statements that specify a condition that must be true at all times in the database. They are used to enforce business rules and data integrity constraints that cannot be enforced using other database mechanisms, such as primary keys, foreign keys, and check constraints. Assertions are evaluated whenever the database is modified, and if the condition specified in the assertion is not satisfied, the modification is rejected. Assertions are checked at the end of a transaction.

Triggers, on the other hand, are procedural code fragments that are executed in response to certain events, such as the insertion, deletion, or modification of data in a table. Triggers can be used to enforce complex business rules that cannot be enforced using other database mechanisms. They are written using SQL or other programming languages, and they can access the data in the database and modify it if necessary. Triggers can be executed before or after an event, and they can be defined to execute either once or multiple times.

The main differences between assertions and triggers are:

- **Definition:** Assertions are declarative statements that define a condition that must be satisfied, whereas triggers are procedural code fragments that execute in response to an event.

- **Purpose:** Assertions are used to enforce data integrity constraints and business rules that cannot be enforced using other database mechanisms, while triggers are used to enforce complex business rules that cannot be enforced using other database mechanisms.
- **Evaluation time:** Assertions are evaluated at the end of a transaction, while triggers are executed in response to an event.
- **Language:** Assertions are written using SQL, while triggers can be written using SQL or other programming languages.

11. Consider following bank database:
Branch-schema = (branch-name, branch-city, assets)
Loan-schema = (loan-number, branch-name, amount)
Write an assertion for the bank database to ensure that the Assets value for the Koteshwor branch is equal to the sum of all the amounts lent by the Koteshwor branch. [2018 Fall]

ANS:
```
CREATE ASSERTION assets_equal_to_sum_of_lent_amounts
CHECK (
(SELECT SUM(amount) FROM Loan WHERE branch_name =
'Koteshwor') =(SELECT assets FROM Branch WHERE
branch_name = 'Koteshwor'));
```

12. Design relational database for the Dept. of Computer Engineering (DoCE) at Pokhara University. Your database should have at least three (3) relations. Describe referential integrity constraint based on the above database of DoCE. [2017 Spring]

ANS: To design a relational database for the Department of Computer Engineering (DoCE) at Pokhara University, we can create following three relations:

Students: This relation will store information about the students enrolled in the department. It can include attributes such as Student ID, Name, Gender, Address, Email, and Year of Enrollment.

Courses: This relation will store information about the courses offered by the department. It can include attributes such as Course ID, Course Name, Course Code and Semester.

Enrollments: This relation will store information about the enrollment of students in courses. It can include attributes such as Student ID (foreign key referencing Students relation), Course ID (foreign key referencing Courses relation), and Enrollment Date.

Referential Integrity Constraint:

To maintain referential integrity in the above database, we can enforce the following constraints:

Foreign Key Constraint (Enrollments table):

The foreign key constraint on the Enrollments table ensures that the Student ID and Course ID values in the Enrollments table reference existing Student ID and Course ID values in the Students and Courses tables, respectively.

This constraint ensures that students can only be enrolled in existing courses offered by the department, and the information in the Enrollments table remains consistent.

Example of Foreign Key Constraint in Enrollments table:

```
CREATE TABLE Enrollments (
EnrollmentID INT PRIMARY KEY,
StudentID INT,
CourseID INT,
EnrollmentDate DATE,
FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);
```

By enforcing this foreign key constraint, any insert or update operation on the Enrollments table will be checked against the referenced tables (Students and Courses) to ensure the existence of the corresponding Student ID and Course ID values. If the referenced values don't exist, the constraint will prevent the operation from being performed, maintaining the referential integrity of the database.

13. Explain the roles of Assertions and Triggers in SQL. When Triggers are not appropriate to use? Give an example. [2015 Fall]

ANS: Database features like assertions and triggers give SQL data manipulation and validation more flexibility and control. Here is a description of the roles they play and an illustration of a situation in which triggers would not be suitable to use:

Assertions

Assertions serve as a means of expressing complex integrity rules that are difficult to enforce using other constraints, such as check constraints or key constraints. The formulation of conditions involving numerous tables and relationships is made possible by them. For instance, an assertion can guarantee that every "Employee" record in the database has a corresponding "Department" record.

Triggers:

Triggers are used to enforce complicated business rules or carry out extra tasks that cannot be completed by constraints alone. They can be used to safeguard data integrity, set up auditing procedures, propagate changes to connected tables, or set off notifications in response to particular circumstances. When a "Employee" record is edited, a trigger can be set up to update a "LastModifiedDate" column.

When Triggers Are Not Appropriate to Use:

Although triggers provide flexibility and extensibility, there are several circumstances in which their use may not be suitable. When complex business logic or extensive processing is required, this is one instance. In general, triggers should be kept quick and light to prevent slowing down activities that modify data. It may be better to handle the logic at the application layer or via stored procedures if it becomes too complex or resource-intensive.

For instance, consider a scenario where an e-commerce system needs to calculate and update order totals based on product prices and quantities whenever an order is placed or modified. If the calculation involves complex pricing rules, discount calculations, or involves interactions with external systems, implementing it solely through triggers may lead to performance issues and make the system harder to maintain. In such cases, it may be more appropriate to handle the calculations and updates in a separate stored procedure or application code, where it can be optimized and managed more effectively.

◆ ◆ ◆