UNIT 5 DATABASE INTEGRITY, FUNCTIONAL DEPENDENCY AND NORMALISATION

Structure **Page Nos.**

5.0 INTRODUCTION

The first block of this course discusses the database concept, relational algebra, Entity relationship model and integrity constraints in the context of relational database design. One of the key aspects of database design is to create relations without any data redundancy.

This unit first explains the concept of entity and referential integrity. It also explains the anomalies in a relational database system. To remove anomalies is through decomposing the database, you need to decompose relations into smaller relations, which are free of those anomalies. This decomposition may be lossless and dependency preserving. The Unit explains the concept of functional dependency, which is the basis of lossless decomposition of a relation into smaller relations.

The unit deals with the standard form of database relations and discusses the process of decomposing relations into different normal forms up to BCNF. The higher normal forms are discussed in the next unit.

5.1 OBJECTIVES

After going through this unit, you should be able to:

- Explain entity integrity and referential integrity constraints of a relation;
- Explain various anomalies that exist in a database system;
- Define the desirable properties of decomposing a relation;
- Define and use functional dependencies to normalise databases.

5.2 DATABASE INTEGRITY

Database integrity refers to maintaining the consistency of data in the database. Data integrity relates to the correctness of data and often is implemented using constraints on attributes in one or more relations. In this section, we will discuss more about two important integrity constraints of a database: the entity integrity constraint and the referential integrity constraint. To define these two, let us once again define the term Key with respect to a Database Management System.

5.2.1 The Keys

Candidate Key: In a relation R, a candidate key for R is a subset of the set of attributes of R, which have the following two properties:

(1)	*Uniqueness*	No two distinct tuples in R have the same value for the candidate key
(2)	*Irreducible*	No proper subset of the candidate key has the uniqueness property.

Every relation must have at least one candidate key which cannot be reduced further. Duplicate tuples are not allowed in relations. Any candidate key can be a composite key also. For Example, (student-id + course-id) together can form the candidate key of a relation called *Result (student-id, course-id, marks)*.

Let us summarise the properties of a candidate key.

- A candidate key must be unique and irreducible.
- A candidate may involve one or more than one attribute. A candidate key that involves more than one attribute is said to be composite.

But why are we interested in candidate keys?

Candidate keys are important because they provide the basic **tuple-level identification** mechanism in a relational system. For example, if the enrolment number is the candidate key of a STUDENT relation, then the answer of the query: "Find student details from the STUDENT relation having enrolment number A0123" will output at most one tuple.

Primary Key

The primary key is the candidate key that is chosen by the database designer as the principal means of identifying entities within an entity set. The remaining candidate keys, if any, are called **alternate keys.**

Foreign Keys

Let us first give you the basic definition of foreign key.

Let R2 be a relation, then a foreign key in R2 is a subset of the set of attributes of R2, such that:

1. There exists a relation R1 (R1 and R2 not necessarily distinct) with a candidate key, and
2. For all time, each value of a foreign key in the current state or instance of R2 is identical to the value of Candidate Key in some tuple in the current state of R1.

The definition above seems to be very heavy. Therefore, let us define it in more practical terms with the help of the following example.

Example 1: Assume that in an organisation, an employee may perform different roles in different projects. Say, XYZ is coding in one project and designing in another. Assume that the information is represented by the organisation in three different relations named EMPLOYEE, PROJECT and ROLE. The ROLE relation describes the different roles required in any project.

Assume that the relational schema for the above three relations are:

EMPLOYEE (EMPID, Name, Designation)
PROJECT (PROJID, Proj_Name, Details)
ROLE (ROLEID, Role_description)

In the relations above EMPID, PROJID and ROLEID are unique and not NULL. As you can clearly observe, you can identify the complete instance of the entity set employee through the attribute EMPID. Thus, EMPID is the primary key of the relation EMPLOYEE. Similarly, PROJID and ROLEID are the primary keys for the relations PROJECT and ROLE respectively.

Let ASSIGNMENT is a relationship between entities EMPLOYEE and PROJECT and ROLE, describing which employee is working on which project and what the role of the employee is in the respective project. Figure 5.1 shows part of E-R diagram for these entities and relationships (for simplicity, no attribute has been shown).

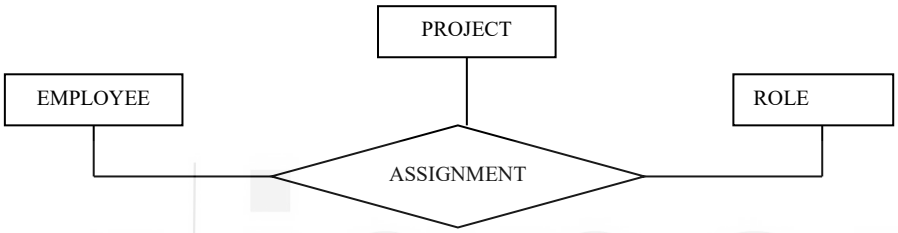


Figure 5.1: E-R diagram for employee role in development

Let us consider sample relation instances as:

EMPLOYEE

EMPID	Name	Designation
101	XYZ	Analyst
102	ABC	Receptionist
103	ARVIND	Manager

PROJECT

PROJID	Proj_name	Details
TCS	Traffic Control System	For traffic shaping.
LG	Load Generator	To simulate load for input in TCS.
B++1	B++_TREE	ISS/R turbo sys

ROLE

ROLEID	Role_descrption
1000	Design
2000	Coding
3000	Marketing

ASSIGNMENT

EMPID	PROJID	ROLEID
101	TCS	1000
101	LG	2000
102	B++1	3000

Figure 5.2: An example relation

You can define the relational scheme for the relation ASSIGNMENT as follows:
ASSIGNMENT (EMPID, PROJID, ROLEID)
Please note now that in the relation ASSIGNMENT (as per the definition of foreign key to be taken as R2) EMPID is the foreign key in ASSIGNMENT relation; it references the relation EMPLOYEE (as per the definition to be taken as R1) where EMPID is the primary key. Similarly, PROJID and ROLEID in the relation ASSIGNMENT are **foreign keys** referencing the relation PROJECT and ROLE respectively.

Now after defining the concept of foreign key, we can proceed to discuss the actual integrity constraints namely Referential Integrity and Entity Integrity.

5.2.2 Referential Integrity

It can be simply defined as:

The database must not contain any unmatched foreign key values.

The term "unmatched foreign key value" means a foreign key value for which there does not exist a **matching value** of the relevant candidate key in the relevant target (referenced) relation. For example, any value existing in the EMPID attribute in ASSIGNMENT relation must exist in the EMPLOYEE relation. That is, the only EMPIDs that can exist in the EMPLOYEE relation are 101, 102 and 103 for the present state/ instance of the database given in *Figure 5.2*. If we want to add a tuple with EMPID value 104 in the ASSIGNMENT relation, it will cause violation of referential integrity constraint. Logically it is obvious, after all the employee 104 does not exist, so how can s/he be assigned any work.

Database modifications can cause violations of referential integrity. We list here the referential action that you may specify for each type of database modification to preserve the referential-integrity constraint:

Delete

During the deletion of a tuple two cases can occur:

Deletion of tuple in relation having the foreign key: In such a case simply delete the desired tuple. For example, in ASSIGNMENT relation you can easily delete the first tuple.

Deletion of the target of a foreign key reference: For example, an attempt to delete an employee tuple in EMPLOYEE relation whose EMPID is 101. This employee appears not only in the EMPLOYEE but also in the ASSIGNMENT relation. Can this tuple be deleted? If you delete the tuple in EMPLOYEE relation then two unmatched tuples are left in the ASSIGNMENT relation, thus causing violation of referential integrity constraint. Thus, the following two choices exist for such deletion:

RESTRICT – The delete operation is "restricted" to only the case where there are no matching tuples in the referencing relation. For example, you can delete the EMPLOYEE record of EMPID 103 as no matching tuple in ASSIGNMENT but not the record of EMPID 101.

CASCADE – The delete operation "cascades" to delete those matching tuples also. For example, if the delete mode is CASCADE, then deleting an employee data having EMPID as 101 from EMPLOYEE relation will also cause deletion of 2 more tuples from ASSIGNMENT relation.

Insert

The insertion of a tuple in the target of reference does not cause any violation. However, insertion of a tuple in the relation in which we have the foreign key, for example, in ASSIGNMENT relation it needs to be ensured that all matching target candidate key exist; otherwise, the insert operation can be rejected. For example, one of the possible ASSIGNMENT insert operations would be (103, LG, 3000).

Modify

Modifying or updating operations changes the existing values. If these operations change the value that is the foreign key also, the only check required is the same as that of the Insert operation.

What should happen to an attempt to update a candidate key that is the target of a foreign key reference? For example, an attempt to update the PROJID "LG" for which there exists at least one matching ASSIGNMENT tuple? In general, there are the same possibilities as for DELETE operation:

RESTRICT: The update operation is "restricted" to the case where there are no matching ASSIGNMENT tuples. (It is rejected otherwise).

CASCADE – The update operation "cascades" to update the foreign key in those matching ASSIGNMENT tuples also.

5.2.3 Entity Integrity

Before describing the second type of integrity constraint, viz., Entity Integrity, you should be familiar with the concept of **NULL.**

Basically, NULL is intended as a basis for dealing with the problem of missing information. This kind of situation is frequently encountered in the real world. For example, historical records sometimes have entries such as "Date of birth unknown". Hence it is necessary to have some way of dealing with such situations in database systems. Codd proposed an approach to this issue that makes use of special markers called NULL to represent such missing information.

A given attribute in the relation might or might not be allowed to contain NULL. But can the Primary key or any of its components (in case primary key is a composite key) contain a NULL? To answer this question an **Entity Integrity Rule** states:
No component of the primary key of a relation is allowed to accept NULL.
In other words, the definition of every attribute involved in the primary key of any basic relation must explicitly or implicitly include the specifications of NULL NOT ALLOWED.

Foreign Keys and NULL

Let us consider the relations:

DEPT		
DEPT ID	**DNAME**	**BUDGET**
D1	Marketing	10M
D2	Development	12M
D3	Research	5M

EMP			
EMP ID	**ENAME**	**DEPT ID**	**SALARY**
E1	Rohan	D1	40K
E2	Aparna	D1	42K
E3	Ankit	D2	30K
E4	Sangeeta		35K

Suppose that Sangeeta is not assigned any Department. In the EMP tuple corresponding to Sangeeta, therefore, there is no genuine department number that can serve as the appropriate value for the DEPTID foreign key. Thus, one cannot determine DNAME and BUDGET for Sangeeta's department as those values are NULL. This may be a real situation where the person has newly joined and is undergoing training and will be allocated to a department only on completion of the training. Thus, NULL in foreign key values may not be a logical error.

So, the foreign key definition may be redefined to include NULL as an acceptable value in the foreign key for which there is no need to find a matching tuple.

Check Your Progress 1

Consider the following relations:

S

SNO	**SNAME**	**CITY**
S1	Smita	Delhi
S2	Jim	Pune
S3	Ballav	Pune

P

PNO	**PNAME**	**COLOUR**	**CITY**
P1	Nut	Red	Delhi
P2	Bolt	Blue	Pune
P3	Part1	White	Mumbai

S4	Seema	Delhi
S5	Salim	Agra

P4	Part2	Blue	Delhi
P5	Camera	Brown	Pune
P6	Part3	Grey	Delhi

J

JNO	**JNAME**	**CITY**
J1	Sorter	Pune
J2	Display	Bombay
J3	OCR	Agra
J4	Console	Agra
J5	RAID	Delhi
J6	EDP	Udaipur
J7	Tape	Delhi

SPJ

SNO	**PNO**	**JNO**	**QUANTITY**
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J2	400
S2	P2	J7	200
S2	P3	J3	500
S3	P3	J5	400
S4	P4	J3	900

1) For each of the relations, as given above, list the candidate keys. Also, identify the Primary key to each of the relations.

……

……

……

……

2) List the entity integrity constraints, which can be found in relations S, P, J, SPJ? List the domain constraints, if any.

……

……

……

……

3) Which of the relations S, P, J, SPJ has referential constraints? List those constraints.

……

……

……

……

4) For the referential constraints as identified in question 3, suggest suitable referential actions.

……

……

……

……

5.3 REDUNDANCY AND ASSOCIATED PROBLEMS

Let us consider the following relation STUDENT.

Enrolment Number (StEnroNo)	Student Name (StName)	Student Address (StAddress)	Course Number (CoNo)	Course Name (CoName)	Instructor (CoInstructor)	Office number of the Instructor (InOffice)
050112345	Rohan	D-27, Main Road, Ranchi	MCS–201	Problem Solution	Nayan Kumar	102
050112345	Rohan	D-27, Main Road, Ranchi	MCS–202	Computer Organisation	Anurag Sharma	105
050112345	Rohan	D-27, Main Road, Ranchi	MCS-203	OS	Preeti Anand	103
050111341	Aparna	B-III, Gurgaon	MCS-203	OS	Preeti Anand	103

Figure 5.3: A state of STUDENT relation

The above relation satisfies the properties of a relation and contains a single value in each cell. Conceptually it is convenient to have all the information in one relation, as a single query to the database may produce complete information about a person. Does the student relation, as given in Figure 5.3 has any undesirable characteristics?

You may observe that Figure 5.3 contains duplicate information in several attributes. For example, the student, whose enrolment number is 050112345 has a name - Rohan and the student stays at an address "D-27, Main Road, Ranchi". This information is repetitive in the first three attributes of tuples 1, 2 and 3 (shown in Figure 5.3 in red colour). Similarly, the information that course name for number MCS-203 is OS and it is taught by "Preeti Anand", whose office number is 103 (shown in Figure 5.3 in purple colour). You can observe that even this information is repetitive in tuple 3 and tuple 4. Thus, the relation of Figure 5.3 has the undesirable **Data Redundancy**.

In addition, when a new student takes admission and enrolls for the course MCS-203, then the entire information about the MCS-203 will be added to the relation. Such repetitive information not only increases the size of the database, but also results in several problems, which are discussed below.

1. *Update Anomaly*: Consider the data redundancy of student name and address, as shown in Figure 5.3. Assume that the student Rohan shifts from Ranchi to New Delhi at a new address "F-102, Maidan Garhi, New Delhi". This will require updating the address of Rohan in all the three tuples of the Student relation. All the three tuples must be updated consistently. If this does not happen, then the address of Rohan will be inconsistent. This inconsistency is the result of update operation on redundant data. Thus, this anomaly is termed an update anomaly, which causes **data inconsistency.**

2. *Insertion Anomaly*: The note that the primary key of the Student relation, as given in Figure 5.3, is composite key consisting of enrolment number and Course Number. Any new tuple to be inserted in the relation must have a value for both the attributes of the primary key, as entity integrity constraint requires that a key may not be totally or partially NULL. However, in the given relation if you want to insert the course number and course name of a new course in the database, it would not be possible until a student enrolls in that course. Similarly, information about a new student cannot be inserted in the database until the student enrolls in a course. These problems are called insertion anomalies.

3. *Deletion Anomalies*: In some instances, useful information may be lost when a tuple is deleted. For example, if you delete the tuple corresponding to student 050111341 enrolled for MCS-203, you will lose relevant information about the

student viz. enrolment number, name and address of this student. Similarly, deletion of tuple having Student Name "Rohan" and Course Number 'MCS-202" will result in loss of information that MCS-202 is named "Computer Organisation" having an instructor "Anurag Sharma", whose office number is 105. This is called deletion anomaly.

The anomalies arise primarily because the relation STUDENT has information about students as well as courses. One solution to the problems is to decompose the relation into two or more smaller relations, so that a relation contains information about one thing. But what should be the basis of this decomposition? To answer the questions let us try to articulate dependence of data within a relation with the help of the following *Figure 5.4*:

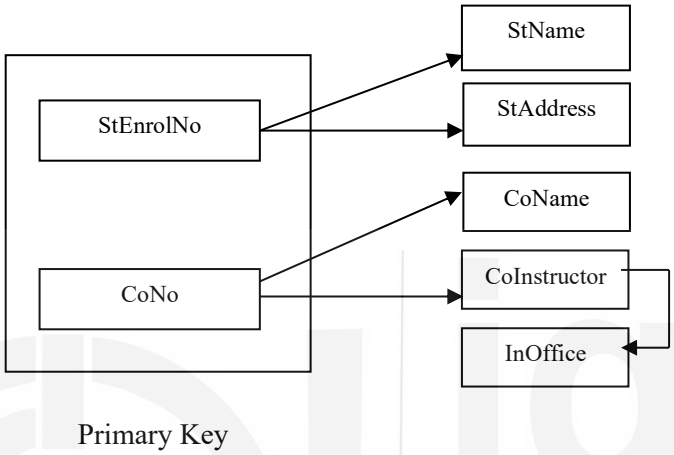


Figure 5.4: The dependencies of relation in Figure 5.3

In *Figure 5.4*, an arrow shows dependence of data attributes. For example, you may notice that two arrows (links) are originating from an attribute student enrolment number (*StEnrolNo*). One of these links is to student name attribute (*StName*) and the second link to student address (*StAddress*). The link between attribute *StEnrolNo* and student name attribute (*StName*) denotes that enrolment number attribute uniquely determines the student's name. For example, in Figure 5.3 the enrolment number 050112345 uniquely determines that name of the student is Rohan. Likewise, you may determine the dependence among different attributes. You may notice that the dependence of data can exist even among the attributes which are not the part of the primary key, such as, a dependence from course instructor (*CoInstructor*) attribute to instructor office number (*InOffice*) attribute. The dependence among the attributes forms the basis of decomposition of a relation into two or more relations, this is called the normalisation. The objective of this decomposition is to reduce the three anomalies in the decomposed relations. However, normalisation should not result in loss of information, which means that you should be able to obtain the original relation's information by taking JOIN of the relations obtained after decomposition.

A relation that needs to be normalised may have a very large number of attributes. In such relations, it is almost impossible for a person to conceptualise all the information and suggest a suitable decomposition to overcome the problems. Such relations need an algorithmic approach for finding if there are problems in a proposed database design and how to eliminate them if they exist. The discussions of these algorithms are beyond the scope of this Unit, but we will first introduce you to the basic concept that supports the process of Normalisation of large databases. So let us first define the concept of functional dependence in the subsequent section and follow it up with the concepts of normalisation.

5.4 FUNCTIONAL DEPENDENCIES

A database is a collection of related information and it is therefore inevitable that some items of information in the database would depend on some other items of information. The information is either single-valued or multi-valued. The enrolment number of a student and his/her date of birth are single-valued information; qualifications of a person or subjects that an instructor teaches are multi-valued facts. In this section, we will deal with single-valued facts, which forms the basis of the concept of functional dependency. Let us define this concept logically.

Functional Dependency (FD)

Let us consider a single universal relation schema "A". A functional dependency denoted by X → Y, between two sets of attributes X and Y that are subset of universal relation "A" specifies a constraint on the possible tuples that can form a relational state of "A". Consider any two tuples of a relation A, say **t1** and **t2,** a FD is said to exist between two sets of attributes X to Y, if the following holds in A:

If **t1**(X) = **t2**(X), then **t1**(Y) = **t2**(Y) must be true.

It means that, if tuple **t1** and tuple **t2** have same values for attributes X, then to hold X→Y on "A", **t1** and **t2** must have same values for attributes Y also.

Thus, FD X→Y means that the values of the Y component of a tuple in "A" depend on or is determined by the values of X component. In other words, the value of the Y component is uniquely determined by the value of the X component. This is functional dependency from X to Y (**but not Y to X**) that is, Y is functionally dependent on X.

The relation schema "A" determines the function dependency of Y on X (X→Y) when and only when:

1) if tuples in "A", which agree on their X value, then
2) they **must** agree on their Y values too.

Please note that if X →Y in "A", does not mean Y→ X in "A".

Please note that functional dependency (FD) does not imply a one-to-one relationship between X *and* Y.

For example, the functional dependencies of *Figure 5.4* are:

StEnrolNo	→	StName, StAddress
CoNo	→	CoName, CoInstructor
CoInstructor	→	InOffice

These functional dependencies imply that there can be only one student name for each **StEnrolNo**, only one address for each student and only one course name for each **CoNo**. Please note, given this set of FDs, it is possible that two or more students, who have different enrolment numbers may have the same name. In addition, two or more students can reside at the same address. However, two different students cannot have the same enrolment number.

Similarly, consider **CoNo → CoInstructor**, the dependency implies that no subject can have more than one instructor (perhaps this is not a very realistic assumption). Functional dependencies therefore place constraints on what information the database may store. In addition, in the example above, you may be wondering if the following FDs hold:

StName → **StEnrolNo** **(1)**

CoName → CoNo **(2)**

Certainly, there is nothing in the given instance of the database relation presented that contradicts the functional dependencies as above. However, whether these FDs hold or not would depend on whether the university or college, whose database you are considering, allows two different students to have the same name and two different courses to have the same course names. If it was the enterprise policy to have unique course names, then **(2)** holds. If two students have exactly the same name, then (**1)** does not hold.

Let us use an E-R diagram to show various FDs (Please refer to *Figure 5.5*). A simple example of functional dependency in this ERD is when X is a primary key of an entity (e.g., enrolment number) and Y is some single-valued property or attribute of the entity (e.g., student name). **X → Y** then must always hold. (Why?)

Functional dependencies also arise in relationships. Let C and D be the primary keys of two strong entity sets participating in a relationship. If the relationship is one-to-one, both the FDs **C → D** and **D → C** will hold. If the relationship is many-to-one (C on many side), an FD **C → D** will hold but the FD **D → C** will NOT hold. In case, a relationship cardinality is many-to-many, then the key attributes of the participating entities would not have any functional dependency between them.

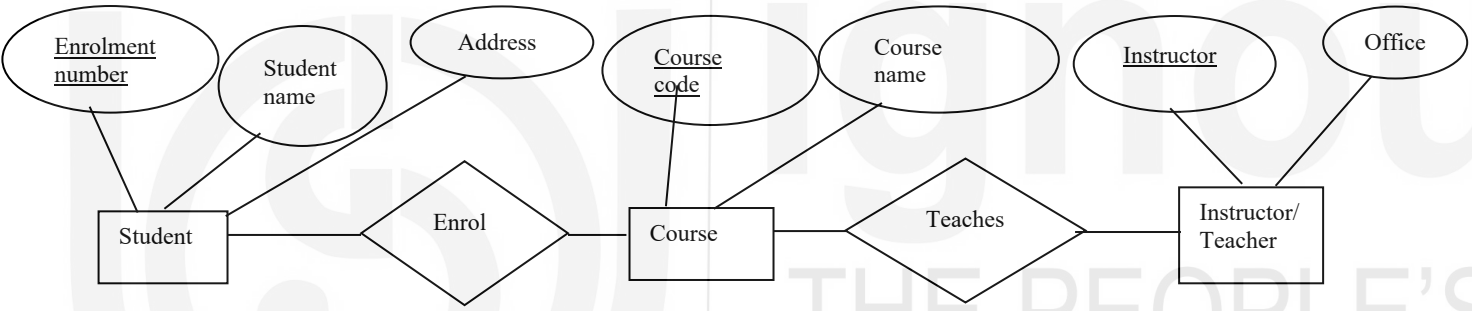


Figure 5.5: ERD of Entities – Student, Course and Teacher

ER diagram Figure 5.5 can be converted to the following relations:

Relations of Entities

STUDENT (StEnrolNo, StName, StAddress)

FDs: StEnrolNo → StName, StAddress

COURSE (CoNo, CoName)

FD: CoNo → CoName

INSTRUCTOR (CoInstructor, InOffice) //CoInstructor is instructor Id

FD: inId → InOffice

Relations of Relationships

ENROL (StEnrolNo, CoCode) (assuming many-to-many cardinality)

FD: StEnrolNo, CoNo → NULL

Assuming that one course can be taught by only one teacher, but one teacher can teach many courses, you need to redesign COURSE relation as:

COURSE (CoNo, CoName, CoInstructor)

FDs: CoNo → CoName, CoInstructor

Identification of FDs:

Identification of FDs, in general, is not trivial. You need to study the domain of attributes and relationships among these attributes. You cannot identify FDs by using a set of rules. The following example explains this.

Consider the following relation:

STUDENT-COURSE (enrolno, sname, cname, classlocation, hours)

The following functional dependencies may exist on this relation (you should identify the FDs primarily from constraints, there is no thumb rule to do so otherwise):

- **enrolno → sname** (the enrolment number of a student uniquely determines the student names alternatively; you can say that sname is functionally determined/dependent on enrolment number).
- **classcode → cname, classlocation,** (the value of a class code uniquely determines the class name and class location.
- **enrolno, classcode → Hours** (a combination of enrolment number and class code values uniquely determines the number of hours and students' study in the class per week (Hours).

The semantic property of functional dependency explains how the attributes in "A" are related to one another. A FD in "A" must be used to specify constraints on its attributes that must hold at all times.

For example, a FD (State, City, Place) → Pin_code should hold for any address in India. It is also possible that certain functional dependencies may cease to exist in the real world if the relationship changes, for example, the FD Pin_code → Area_code used to exist as a relationship between postal codes and telephone number area codes in India, however. with the proliferation of mobile telephone, the FD is no longer true.

The set of FDs over a relation can be optimised to obtain a minimal set of FDs called the canonical cover. However, these topics are beyond the scope of this course and can be studied by consulting further readings.

5.5 NORMALISATION USING FUNCTIONAL DEPENDENCIES

Codd in the year 1972 presented three normal forms (1NF, 2NF, and 3NF). These were based on functional dependencies among the attributes of a relation. Later Boyce and Codd proposed another normal form called the Boyce-Codd normal form (BCNF). The fourth and fifth normal forms are based on multi-valued dependency and join dependencies and were proposed later. In this section we will cover normal forms till BCNF only. Fourth and fifth normal forms are discussed in the next unit. For all practical purposes, 3NF or the BCNF are quite adequate since they remove the anomalies discussed for most common situations. It should be clearly understood that there is no obligation to normalise relations to the highest possible level. Performance should be taken into account and sometimes an organisation may take a decision not to normalise, say, beyond third normal form. But it should be noted that such designs should be careful enough to take care of anomalies that would result because of the decision above.

Intuitively, the second and third normal forms are designed to result in relations such that each relation contains information about only one thing (either an entity or a relationship). A sound E-R model of the database would ensure that all relations either provide facts about an entity or about a relationship resulting in the relations that are obtained being in 2NF or 3NF.

The normalisation of a relation till BCNF is established using the FDs. The normalisation process depends on the assumptions that:

1) a set of functional dependencies is given for each relation, and
2) Each relation has a designated primary key.

The normalisation process proceeds in a top-down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as found necessary during analysis. Therefore, normalisation upto BCNF is looked upon as a process of analysing the given relation schemas based on their condition (FDs and Primary Keys) to achieve the desirable properties:

- firstly, minimising redundancy, and
- secondly minimising the insertion, deletion and update anomalies.

Thus, the normalisation provides the database designer with:

1) a formal framework for analysing relation schemas.
2) a series of normal form tests that can be normalised to any desired degree.

Decomposition through normalisation should include any or both of the following properties.

1) the lossless join and non-additive join property, and
2) the dependency preservation property.

Do you always normalise the database to the highest normal form? Due to performance reasons, database designers sometimes leave relations in lower normalisation forms. This is called denormalisation.

Let us now define the normal forms in more detail.

5.5.1 The First Normal Form (1NF)

Let us first define 1NF:

Definition: A relation (table)R is in 1NF if every attribute of R takes atomic values. In other words, the following conditions hold in R:

1. *There are no duplicate rows or tuples in the relation.*
2. *Each data value stored in the relation is single-valued.*
3. *Entries in a column (attribute) are of the same kind (type).*

Please note that in a 1NF relation, the order of the tuples (rows) and attributes (columns) does not matter.

The first requirement above means that the relation **must have a key**. The key may be single attribute or composite key. It may even, possibly, contain all the columns. The first normal form defines only the basic structure of the relation and does not resolve the anomalies discussed in Section 5.3.

The relation STUDENT (StEnrolNo, StName, StAddress, CoNo, CoName, CoInstructor, InOffice) of *Figure 5.3* is in 1NF. The primary key of the relation is a composite key of attributes StEnrolNo and CoNo.

5.5.2 The Second Normal Form (2NF)

The relation of *Figure 5.3* is in 1NF, yet it suffers from all the anomalies. Therefore, a second normal form may be defined to overcome the anomalies.

Definition: A relation is in Second Normal Form (2NF) if it fulfills the following criteria (assuming the relation has only one candidate key, which is selected as the primary key of the relation):

(i) The relation fulfills the criteria of the First Normal Form (1NF), and

(ii) All those attributes, which are not part of any primary key, are fully functionally dependent on the primary keys.

Key features of 2NF:

- The partial dependency will exist, only if the primary key is composite. Therefore, if the primary key of a relation consists of a single attribute, then the given 1NF relation would be in 2NF also.
- This normal form decomposes a relation so that relations store information about one thing.

Please refer to *Figure 5.4,* which illustrates the FDs in the relation STUDENT (StEnrolNo, StName, StAddress, CoNo, CoName, CoInstructor, InOffice). The FDs, as shown in *Figure 5.4* are:

StEnrolNo	→	StName, StAddress	(1)
CoNo	→	CoName, CoInstructor	(2)
CoInstructor	→	InOffice	(3)

Since, from the above FDs, you can create a FD:

StEnrolNo, CoNo → StName, StAddress, CoName, CoInstructor, InOffice

Therefore, the attributes StEnrolNo and CoNo together form the composite key to the relation STUDENT. The relation has only one candidate key, therefore, all the attributes, other than StEnrolNo and CoNo, of the relation are not part of a candidate key. These attributes are also called non-key attributes. The relation STUDENT is in 1NF, but is it in 2NF? No, the FDs at (1) and (2) are clearly violating the fully functionally dependent criteria of 2NF (*Figure 5.4*). Therefore, the relation suffers from the anomalies as shown in Figure 5.3. Next, how will you decompose the STUDENT relation to 2NF relations? You may use FDs of (1), (2) and (3) to do so. FD (1) can be used to create a 2NF relation consisting of these three attributes from STUDENT relation. This part new relation is:

STUDENT1 (StEnrolNo, StName, StAddress)

Similarly, you can use FD (2) to decompose the STUDENT relation further, but what about the attribute 'InOffice'? You find in FD (2) that Course code (CoNo) attribute uniquely determines the name of instructor (refer to FD 2(a)). Also, the FD (3) means that the name of the instructor uniquely determines the office number. This can be written as:

CoNo	→	CoInstructor	(2 (a)) (without CoName)
CoInstructor	→	InOffice	(3)

The above two FDs imply a transitive dependency:

CoNo	→	InOffice	(This is transitive dependency)

The revised FD (2) is:

CoNo	→	CoName, CoInstructor, InOffice	(4)

Use this FD to create another 2NF relation:

COU_INST (CoNo, CoName, CoInstructor, InOffice)

Now, you have the following two 2NF relations, which are obtained by decomposing the STUDENT relation:

STUDENT1 (StEnrolNo, StName, StAddress)
COU_INST (CoNo, CoName, CoInstructor, InOffice)

Are these two 2NF relations sufficient to represent the relations STUDENT? No, as there is no common attribute between the relations. Therefore, they cannot be joined together to get the data of STUDENT relation. You must have a relation that joins the two decomposed relations. This relation would have the primary key attributes of the STUDENT relation and any other attribute that is fully

functionally dependent on this primary key. However, there is no attribute in STUDENT relation that is fully dependent on the composite primary key. Thus, you will create a third relation using the composite primary key of the STUDENT relation, as shown below:

COURSE_STUDENT (StEnrolNo, CoNo)

Please note that the COURSE_STUDENT relation can be joined with STUDENT1 relation on StEnrolNo attribute and with COU_INST relation on CoNo attribute.

The relation STUDENT in 2NF form would be:

STUDENT1 (StEnrolNo, StName, StAddress)	2NF(a)
COU_INST (CoNo, CoName, CoInstructor, InOffice)	2NF(b)
COURSE_STUDENT (StEnrolNo, CoNo)	2NF(c)

5.5.3 The Third Normal Form (3NF)

Although, transforming a relation that is not in 2NF into a number of relations that are in 2NF removes many of the anomalies, it does not necessarily remove all anomalies. Thus, further Normalisation is sometimes needed to ensure further removal of anomalies. These anomalies arise because a 2NF relation may have attributes that are not directly related to the primary key of the relation.

*Definition: A relation is in third normal form if it is in 2NF and every non-key attribute of the relation is non-transitively dependent on the primary key of the relation***.**

But what is **non-transitive** dependence?
Let A, B and C be three attributes of a relation R such that **A → B** and **B → C**. From these FDs, we may derive **A → C**. This dependence **A → C** is transitive.

Now, let us reconsider the relation 2NF (b)

COU_INST (CoNo, CoName, Instruction, InOffice)

Assume that CoName is not unique and therefore CoNo is the only candidate key. The following functional dependencies exist:

CoNo	→	**CoInstructor**	(2 (a))
CoInstructor	→	**InOffice**	(3)
CoNo	→	InOffice	(This is transitive dependency)

You had derived CoNo → InOffice from the functional dependencies 2(a) and (3) for decomposition to 2NF. The relation is, however, not in 3NF since the attribute 'InOffice' is not directly dependent on attribute 'CoNo' but is transitively dependent on it and should, therefore, be decomposed as it has all the anomalies. The primary difficulty in the relation above is that an instructor might be responsible for several subjects, requiring one tuple for each course. Therefore, his/her office number will be repeated in each tuple. This leads to all the problems such as update, insertion, and deletion anomalies. To overcome these difficulties, you need to decompose the relation 2NF(b) into the following two relations:

COURSE (CoNo, CoName, CoInstructor)
INST (CoInstructor, InOffice)

Please note these two relations and 2NF (a) and 2NF (c) are already in 3NF. Thus, the relation STUDENT in 3 NF would be:

STUDENT1 (StEnrolNo, StName, StAddress)
COURSE (CoNo, CoName, CoInstructor)
INST (CoInstructor, InOffice)
COURSE_STUDENT (StEnrolNo, CoNo)

The 3NF is usually quite adequate for most relational database designs. There are, however, some situations where a relation may be in 3 NF but have anomalies. For example, consider a relation STUDENT5 (StEnrolNo, StName, CoNo, CoName). Assume it has the following set of FDs:

StEnrolNo → StName
StName → StEnrolNo
CoNo → CoName
CoName → CoNo

Is the relation STUDENT5 is in 3NF? The FDs of this relation can be written as:

StEnrolNo, CoNO → StName, CoName
StEnrolNo, CoName → StName, CoNO
StName, CoNO → StEnrolNo, CoName
StName, CoName → StEnrolNo, CoNO

Therefore, the relation STUDENT5 has the following candidate keys.

(StEnrolNo, CoNo)
(StEnrolNo, CoName)
(StName, CoNo)
(StName, CoName)

Figure 5.6 shows the functional dependency diagram for this relation.

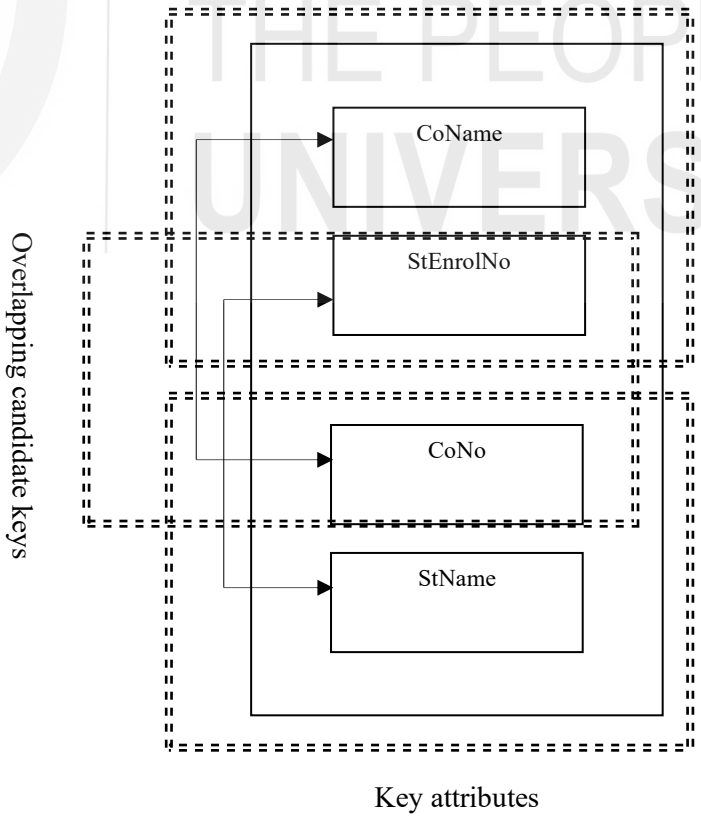


Figure 5.6: Functional Diagram for STUDENT5 relation

You may observe that all the attributes of STUDENT5 relations are prime attributes as they are part of some candidate key. You may also observe that the relation has overlapping candidate keys. Therefore, no non-key attribute exists in the relation. Hence, the relation follows the criteria of 2NF and 3NF and is in 3NF. However, this relation still suffers from the three anomalies (please make an instance for the relation), as the attributes in the non-overlapping part of the candidate key can determine each other. Therefore, the relation is to be normalised into a stronger normal form called BCNF.

5.5.4 Boyce-Codd Normal Form (BCNF)

As stated earlier, the relation STUDENT5 (StEnrolNo, StName, CoNo, CoName) has the following candidate keys:

(StEnrolNo, CoNo) ; (StEnrolNo, CoName)
(StName, CoNo) ; (StName, CoName)

Since the relation has no non-key attributes, the relation is in 2NF and also in 3NF. However, the relation suffers from anomalies (please check it yourself by making the relational instance of the STUDENT5 relation).

The difficulty in this relation is being caused by dependence within the attributes of composite candidate keys.

Definition: *A relation is in Boyce-Codd Normal Form (BCNF) if it fulfills the following criteria:*

(i) The relation fulfills the criteria of the Third Normal Form (3NF), and

(ii) All the determinants (the left side of an FD) are the candidate key.

A relation that is in 3NF and does not have overlapping candidate keys, will also be in BCNF. However, if a 3NF relation have following features then it is NOT in BCNF:

(a) It has overlapping composite candidate keys.
(b) The non-overlapping attributes of two candidate keys are functionally dependent.

For example, in the relation STUDENT5, the candidate keys:
(StEnrolNo, CoName) and (StName, CoName) have non-overlapping attributes StEnrolNo and StName. Further, StEnrolNo → StName
The relation STUDENT5 (StEnrolNo, StName, CoNo, CoName) is in 3NF, but not in BCNF.
You can decompose the relation into the following relations using FD StEnrolNo → StName as:
STUDENT6(StEnrolNo, CoNo, CoName)
STUDENTNAME (StEnrolNo, StName)
STUDENT6 is still not in BCNF as it has overlapping candidate keys: (StEnrolNo, CoNo) and (StEnrolNo, CoName) and the FD CoNo → CoName
Thus, STUDENT6 can be decomposed using the FD as above to relations:
STUDENT7 (StEnrolNo, CoNo)
STUDENTNAME (StEnrolNo, StName)
COURSENAME (CoNo, CoName)

The following is another example of BCNF. Consider the relation:
ENROL (StEnrolNo, StName, CoNo, CoName, Dateenrolled)
Let us assume that the relation has the following FDs (We have assumed that CoName are unique):

StEnrolNo → StName
CoNo → CoName

CoName → CoNo
StEnrolNo, CoNo → Dateenrolled

The candidate keys of the relation would be:

(StEnrolNo, CoNo)
(StEnrolNo, CoName)

Due to dependency between the non-overlapping attributes of the candidate keys, the 3NF relation ENROL is NOT in BCNF. This relation will have all the stated anomalies (Please draw the relational instance and check these problems). The BCNF decomposition of the relation would be:

STUD1 (StEnrolNo, StName)
COU1 (CoNo, CoName)
ENROL1 (StEnrolNo, CoNo, Dateenrolled)

You now have a relation that only has information about students, another only about subjects and the third only about relationship enrolls.

Higher Normal Forms:

Are there more normal forms beyond BCNF? Yes, however, these normal forms are not based on the concept of functional dependence. Further normalisation is needed if the relation has multi-valued, join dependencies, etc. These are discussed in Unit 6.

Check Your Progress 2

1) Given the following relation of book issue and return in a Library:
BOOKISSUERETURN (student_id, student_name, bookID, book_title, issuedOn, returnedOn)
A student can get many books issued to him/her. Explain anomalies in the relation above with the help of a relational instance.
……
……

2) Identify the functional dependencies in the relation given in question 1. What are the candidate keys and which of these can be a primary key?
……
……

3) Normalise the relation of problem 1.
……
……
……
……

5.6 DESIRABLE PROPERTIES OF DECOMPOSITION

In the previous section, you have learnt the process of normalisation using functional dependency. Normalizing a relation entails decomposing a relation into a number of non-disjoint projections of the relation based on the FDs, so that the three anomalies are minimised. But why are these projections non-disjoint? The non-disjoint relations allow reconstruction of the original relation instance through JOIN operation. In this section, we discuss about the properties of the a good decomposition.

The decomposition of a relation should fulfill the following:

Attribute Preservation: All the attributes of the relation, which is being decomposed, should be part of at least one of the decomposed relations. This is a necessary condition, otherwise the decomposition will be lossy.

Lossless-Join Decomposition: For explaining the concept of lossless-join decomposition, let us first explain a lossy decomposition with the help of an example. This example also explains, why FDs should be used to perform proper decomposition of a relation.

Example: Consider the following instance of a STUDENT9 relation.

STUDENT9 (StEnrolNo, CoNo, Dateenrolled, CoInstructor, InRoom)

With the following set of FDs:

StEnrolNo, CoNo → Dateenrolled
CoNo → CoInstructor
CoInstructor → InRoom

Suppose you decompose the STUDENT9 relation randomly into two relations ST1 and ST2 as follows:

ST1 (StEnrolNo, CoNo, Dateenrolled, CoInstructor)
ST2 (StEnrolNo, InRoom)

Are there problems with this decomposition? Yes, but for the time being, let us focus on the question, whether this decomposition is lossless? Consider the following instance of the relation:

STUDENT9

StEnrolNo	CoNo	Dateenrolled	CoInstructor	InRoom
1001	MCS-201	01-02-2022	Preeti	1
1001	MCS-202	01-02-2022	Salim	2
1002	MCS-201	13-02-2022	Preeti	1
1002	MCS-203	13-02-2022	Shashi	3
1003	MCS-202	15-02-2022	Salim	2

Figure 5.7: An instance of STUDENT9 relation

The decomposed relations ST1 and ST2 would be:

ST1

StEnrolNo	CoNo	Dateenrolled	CoInstructor
1001	MCS-201	01-02-2022	Preeti
1001	MCS-202	01-02-2022	Salim
1002	MCS-201	13-02-2022	Preeti
1002	MCS-203	13-02-2022	Shashi
1003	MCS-202	15-02-2022	Salim

ST2

StEnrolNo	InRoom
1001	1
1001	2
1002	1
1002	3
1003	2

Will you be able to reconstruct the original instance of relation using ST1 and ST2? For this simply take a JOIN of the two relations ST1 and ST2 on StEnrolNo, which is the only common attribute. The joined instance would be:

ST1JOINST2

StEnrolNo	CoNo	Dateenrolled	CoInstructor	InRoom
1001	MCS-201	01-02-2022	Preeti	1
1001	MCS-201	01-02-2022	Preeti	2
1001	MCS-202	01-02-2022	Salim	1
1001	MCS-202	01-02-2022	Salim	2
1002	MCS-201	13-02-2022	Preeti	1
1002	MCS-201	13-02-2022	Preeti	3
1002	MCS-203	13-02-2022	Shashi	1

1002	MCS-203	13-02-2022	Shashi	3
1003	MCS-202	15-02-2022	Salim	2

Thus, the resulting relation obtained is not the same as that of *Figure 5.7*. The joined relation contains a number of spurious tuples that were not in the original relation. Because of these additional tuples, you have lost the information, such as the instructor Preeti is located in Room number 1. Such decompositions are called **lossy decompositions**. A lossless join decomposition guarantees that the join will result in exactly the same relation as was decomposed. One might think that there might be other ways of recovering the original relation from the decomposed relations but, sadly, no other operators can recover the original relation if the join does not (why?). You need to analyse why the decomposition is lossy. The common attribute in the above decompositions was StEnrolNo. The common attribute is the glue that gives us the ability to find the relationships between different relations by joining the relations together. *If the common attribute has been the primary key of at least one of the two decomposed relations, the problem of losing information would not have existed.*
You may use FDs to decompose the relation STUDENT9 into 2NF and then to 3NF, to produce the following relational instance:

STENROL (StEnrolNo, CoNo, Dateenrolled)
using the FD StEnrolNo, CoNo → Dateenrolled

STENROL

StEnrolNo	**CoNo**	**Dateenrolled**
1001	MCS-201	01-02-2022
1001	MCS-202	01-02-2022
1002	MCS-201	13-02-2022
1002	MCS-203	13-02-2022
1003	MCS-202	15-02-2022

COUINST (CoNo, CoInstructor) using the FD CoNo → CoInstructor

COUINST

CoNo	**CoInstructor**
MCS-201	Preeti
MCS-202	Salim
MCS-203	Shashi

INSROOM (CoInstructor, InRoom) using the FD CoInstructor → InRoom

INSROOM

CoInstructor	**InRoom**
Preeti	1
Salim	2

You may please join the three relations to check that that the decomposition is lossless-join decomposition. *The dependency-based decomposition scheme as discussed in the section 5.5 creates lossless decomposition.*

Dependency Preservation: It is clear that the decomposition must be lossless so that you do not lose any information from the relation that is decomposed. Dependency preservation is another important requirement since a *dependency is a constraint* on the database. If all the attributes appearing on the left and the right side of a dependency appear in the same relation, then a dependency is considered to be preserved. Thus, dependency preservation can be checked easily. Dependency preservation is important, because as stated earlier, dependency is a constraint on a relation. Thus, if a constraint is split over more than one relation (dependency is not preserved), the constraint would be difficult to meet. We will not discuss this in more detail in this unit. You may refer to the further readings for more details. However, let us state some basic points:

"Normalisation to 3NF is lossless decomposition. It also preserves the dependencies."

"Normalisation to BCNF is lossless decomposition. However, it may not preserve dependencies."

Reduction of Redundancy: *Redundancy is the cause of anomalies. Normalisation results in reduction of redundancy.*

5.6 RULES OF DATA NORMALISATION

In the previous sections, you have gone through various normal forms. In this section, the normalisation using FDs is presented as a set of practical rules:

1. Identify and eliminate Attributes with multiple data values into separate relations.
2. Identify and eliminate those attributes, which are not the part of any key attributes but are dependent on the part of the composite primary key.
3. Identify and eliminate those non-key attributes, which are dependent on other non-key attributes.
4. Identify and eliminate non-overlapping composite candidate key attributes, which are dependent on each other.

Let's explain these steps of Normalisation through an example. Let us make a list of all the employees in the company. In the original employee list, each employee name is followed by any databases that the employee has experience with. Some might know many, and others might not know any.

Employee ID (empid)	Employee Name (empname)	DBMS Known (DBMS)	Department (dept)	Department Location (loc)
101	Gurmeet	MySQL	Quality Assurance	Mumbai
102	Hanif	DB2	Database Design	Delhi
103	Manish	Oracle, PostgreSQL	Frontend Design	Hyderabad
104	Sameer Singh	SQLserver, Oracle	Database Design	Delhi

Figure 5.8: Table of Data Not in 1NF

For attributes Department and Department Location will be considered in Step-3.

5.6.1 Eliminate Repeating Groups

Please observe that Figure 5.8 has a repeating group – DBMS Known. The problem with such repeating groups can be explained with the help of a query. Consider a query - "Find the list of employees, who know Oracle".

To answer this query, you need to perform an awkward scan of the entire list of attributes Database-Known, looking for references to DB2. This is inefficient and an extremely untidy way to retrieve information.

You may convert the relation to 1NF (For the following discussion, we have ignored two attributes - Department and Department Location. These two attributes will be part of Employee relation). The two decomposed relations on eliminating the repeating groups are shown in Figure 5.9. The elimination of repeating group DBMS Known from the Employee relation, which has primary key - Employee ID, leaves the Employee relation in 1NF. However, this elimination requires that you add another relation, which can store information about the DBMS known by each employee. This new relation is named DBMSknown. It has three attributes, including a foreign key for relating the two relations with a JOIN operation. Now, you can answer the question by looking in the database relation for "Oracle" and getting the list of Employees. Please note that although the name of the DBMSs are unique, yet we have added a DBMS code field in the decomposed relation (Figure 5.8):

EMPLOYEE			
empid	**empname**	**dept**	**loc**
101	Gurmeet	Quality Assurance	Mumbai
102	Hanif	Database Design	Delhi
103	Manish	Frontend Design	Hyderabad
104	Sameer Singh	Database Design	Delhi

DBMSknown		
empid	**DBMScode**	**DBMS**
101	1	MySQL
102	2	DB2
103	3	Oracle
103	4	PostgreSQL
104	5	SQLserver
104	3	Oracle

Figure 5.9: 1NF relation after Eliminating Repeating Groups

5.6.2 Eliminate Redundant Data

In the "*DBMSknown*" relation above, the primary key is made up of the *empid* and the *DBMScode*. The attribute *DBMS* depends only on the *DBMScode*. The same *DBMS* will appear redundantly every time its associated *DBMScode* appears in the *DBMSknown* Relation. Thus, this database relation has redundancy, for example, *DBMScode* value 3 is Oracle, which is repeated twice. In addition, it also suffers insertion anomaly that is you cannot enter Sybase in the relation as no employee has that database skill.

The deletion anomaly also exists. For example, if you remove an employee with *empid* 3; no employee has a skill in PostgreSQL and the information that *DBMScode* 4 is the code of PostgreSQL will be lost.

To avoid these problems, you need a second normal form. To achieve this, you isolate the attributes that depend on the entire composite key from the attributes that depend only on the *DBMScode*. This results in decomposition of *DBMSknown* relation into two relations: "DBMSlist" and "EMPDBMS" which lists the databases for each employee. The EMPLOYEE relation is already in 2NF as all the *empid* determines all other attributes.

EMPDBMS	
empid	**DBMScode**
101	1
102	2

DBMSlist	
DBMScode	**DBMS**
1	MySQL
2	DB2

103	3
103	4
104	5
104	3

3	Oracle
4	PostgreSQL
5	SQLserver

5.6.3 Eliminate Columns Not Dependent on Key

The Employee Relation satisfies -

First normal form - As it contains no repeating groups.

Second normal form - As it does not have a multi-attribute key.

Now, let us add the remaining two attributes of Employee relation. The employee relation is in 2NF but not 3NF. Why?

EMPLOYEE			
empid	**empname**	**dept**	**loc**
101	Gurmeet	Quality Assurance	Mumbai
102	Hanif	Database Design	Delhi
103	Manish	Frontend Design	Hyderabad
104	Sameer Singh	Database Design	Delhi

The key to the Employee relation is *empid*. The attribute *loc* describes information about the Department and not about an Employee. To achieve the third normal form, *dept* and *loc* must be moved into a separate relation. Since they describe a department, thus, the attribute *dept* becomes the key of the new "Department" relation.
The motivation for this decomposition is that you want to avoid update, insertion and deletion anomalies.

EMPLOYEElist		
empid	**empname**	**dept**
101	Gurmeet	Quality Assurance
102	Hanif	Database Design
103	Manish	Frontend Design
104	Sameer Singh	Database Design

DEPARTMENT Relation		
deptid	**dept**	**loc**
1	Quality Assurance	Mumbai
2	Database Design	Delhi
3	Frontend Design	Hyderabad

You may use these steps for decomposition till BCNF.

Check Your Progress 3

1) Why should functional dependencies be preserved in decomposition?
……
……
……
……
……

2) Explain the term lossless-join decomposition.
……
……
……
……

3) List various steps that may be followed to decompose a relation up to BCNF.

……

……

……

……

5.7 SUMMARY

This unit discusses some of the most important aspects of a good database design. The unit first defines the concept of referential integrity constraint and entity integrity constraints. It then discusses different forms of normalisation based on the concept of function dependency. A functional dependency highlights the relationships among the attributes of a relation. Different dependency among attributes leads to the problems of update anomalies, insertion anomalies and deletion anomalies. Different forms of normalisation decompose relations, using the FDs, to remove anomalies. The concept of FD is used for the process of normalisation.

This unit also defines the features of a good decomposition of a relation. The most important being attribute preservation, dependency preservation and lossless-join decomposition. Finally, the unit summarises the rules of normalisation with the help of an example.

5.10 SOLUTIONS/ANSWERS

Check Your Progress 1

1) The following table shows the candidate keys and primary keys of the relations:

Relation	Candidate Keys	Primary key
S (Supplier)	SNO, SNAME*	SNO
P (Part)	PNO, PNAME*	PNO
J (Project)	PROJ NO, JNAME*	PROJNO
SPJ	(SNO+PNO+PROJNO)	(SNO+PNO+PROJNO)

* Only if the values are assumed to be unique, this may be incorrect for large systems.

2) SNO in S, PNO in P, PROJNO in J and (SNO+PNO+PROjNO) in SPJ should not contain NULL value. Also, no part of the primary key of SPJ, that is SNO or PNO or PROJNO should contain NULL value.

3) Foreign keys exist only in SPJ, where SNO references SNO of S; PNO references PNO of P; and PROJNO references PROJNO of J. The referential integrity necessitates that all matching foreign key values must exist.

4) You may select the following referential actions:

   For operations like Delete and update, you should use referential action as RESTRICT. This selection would restrict loss of information from the SPJ relation, in case you delete a tuple in S or P or J relation. Insertion of records does not create a problem provided the referential integrity constraints are met.

Check Your Progress 2

1) The database suffers from all the anomalies; let us demonstrate these with the help of the following relation instance or state of the relation:

student_id	Student_name	bookID	Book_title	issuedOn	returnedOn
A 101	Abishek	0050	DBMS	15/01/05	25/01/05
R 102	Raman	0125	DS	25/01/05	29/01/05
A 101	Abishek	0060	Multimedia	20/01/05	NULL
R 102	Raman	0050	DBMS	28/01/05	NULL

Is there any data redundancy?

Yes, the information is getting repeated about student names and book Title. This may lead to an update anomaly in case of changes made to the data value of the book. In addition, the library may be having many more books that have not been issued yet. The information of such books cannot be added to the relation as the primary key to the relation is: (student_id + bookID + issuedOn). (This would involve the assumption that the same book can be issued to the same student only once in a day). Thus, you cannot enter the information about bookID and book_title of those books, which has not been issued to a student. This is an insertion anomaly. Similarly, you cannot enter student_id if a book is not issued to that student. This is also an insertion anomaly. As far as the deletion anomaly is concerned, suppose Abishek did not collect the Multimedia book, so this record needs to be deleted from the relation (tuple 3). This deletion will also remove the information about the Multimedia book that is its bookID and book_title. This is a deletion anomaly for the given instance.

2) The FDs of the relation are:

student_id → student_name (1)
bookID → book_title (2)
bookID, student_id, issuedOn → returnedOn (3)

Why is the attribute issuedOn on the left hand of the FD above? Because a student, for example, Abishek can be issued the book having the bookID 0050 again on a later date, let say in February after Raman has returned it. Also note that returnedOn may have NULL value, but that is allowed in the FD. FD only necessitates that the value may be NULL or any specific data that is consistent across the instance of the relation.

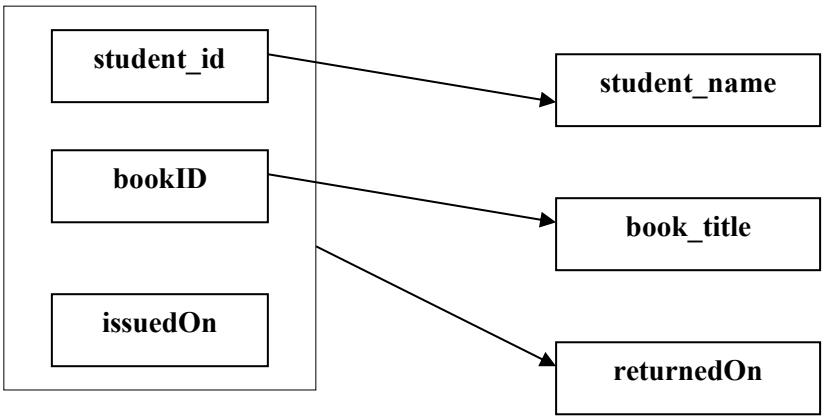
Just one candidate key, which is also chosen as the primary key, is: bookID, student_id, issuedOn.

Some interesting domain and procedural constraints in this database are:

- A book cannot be issued again unless it is returned.
- A book can be returned only after the date on which it has been issued.
- A student can be issued a limited/maximum number of books at a time.

You will study about these constraints later in the Block.

3) The relation is in 1NF. The following is a FD diagram for the relation:



1NF to 2NF:

The composite key to the relation is (student_id + bookID). The attributes student_name depends on student_id, which is part of the composite key, likewise book_title attributes is dependent on bookID attribute, which is part of the composite key so the decomposition based on following FDs would be:

MEMBER (student_id, student_name) [Reason: FD (1)]
BOOK (bookID, book_name) [Reason: FD (2)]
ISSUE_RETURN (student_id, bookID, issuedOn, returnedOn)
[Reason: FD (3)]

2NF to 3 NF and BCNF:

All the relations are in 3NF and BCNF also. As there is no dependence among non-key attributes and there are no overlapping candidate keys.

Please note that the decomposed relations have no anomalies. Let us map the relation instance here:

MEMBER

Member - ID	Members Name
A 101	Abhishek
R 102	Raman

BOOK

Book - Code	Book – Name
0050	DBMS
0060	Multimedia
0125	OS

ISSUE_RETURN

Member - ID	Book - Code	Issue - Date	Return – Date
A 101	0050	15/01/05	25/01/05
R 102	0125	25/01/05	29/01/05
A 101	0060	20/01/05	NULL
R 102	0050	28/01/05	NULL

i) There is no redundancy, so no update anomaly is possible in the relations above.

ii) The insertion of new book and new student can be done in the BOOK and MEMBER tables respectively without any issue of book to student or vice versa. So, no insertion anomaly.

iii) Even on deleting record 3 from ISSUE_RETURN it will not delete the information the book 0060 titled as Multimedia as this information is in separate table. So, no deletion anomaly.

Check Your Progress 3

1) Dependency preservation within a relation helps in enforcing constraints that are implied by dependency over a single relation. In case, you do not preserve dependency then constraints might be enforced on more than one relation that is quite troublesome and time consuming.
2) A lossless join decomposition is one in which you can reconstruct the original table without loss of information that means exactly the same tuples are obtained on taking join of the relations obtained on decomposition. Lossless join decomposition requires that decomposition should be carried out on the basis of FDs.

3) The steps of Normalisation are:

    1. Remove repeating groups of each of the multi-valued attributes.
    2. Then remove redundant data and its dependence on part keys.
    3. Remove columns from the table that are not dependent on the key, that is remove transitive dependency.
    4. Check if there are overlapping composite candidate keys. If yes, check for any dependency among the non-overlapping attributes of the composite candidate keys; and remove such dependency.

UNIT 6 HIGHER NORMAL FORMS

Structure

6.0 INTRODUCTION

In the previous unit of this block, you have gone through the concept of single-valued dependency, called functional dependency (FD). Further, the previous Unit discussed different forms of normalisations that can be arrived at by removing different types of single-valued dependency, viz. 1NF, 2NF, 3NF and BCNF. However, relational databases may have several other types of dependencies, which results in redundancy in a relation. Such dependencies, thus causes the update anomaly, insertion anomaly and deletion anomaly in a relation. Therefore, more normal forms have been designed to address this problem.

This Unit focuses on the higher normal forms, namely fourth normal form (4NF), which is based on the concept of multivalued dependency (MVD); and fifth normal form (5NF), which is based on the concept of Join dependency. The unit also introduces you to other normal forms. For more details on other normal forms, you may refer to the further readings. In general, these higher normal forms are not very popular among industries, however they propose good theoretical perspectives for the database design.

6.1 OBJECTIVES

After going through this unit, you should be able to:

- Explain the concept of multivalued dependencies;
- Perform normalisation up to 4NF;
- Explain the join dependency;
- Explain the 5NF;
- Define other normal forms.

6.2 MULTIVALUED DEPENDENCIES

An attribute in a relational model represents only a single value information. How will you represent the information if a particular attribute is multivalued? Such multivalued attributes would require that

information of all other attributes is repeated in an instance of the relation. This will result in multiple tuples in a single relation to represent information about one entity. The primary key of such a relation would be a composite key involving the primary key of the entity and the multivalued attribute. This situation becomes worse, if an entity has more than one multivalued attribute. Such an entity will be represented by several tuples. The following example explains this situation:

Example 1: Let us consider a relation 'employee'.

emp (e#, project, tool)

An employee can work on several projects and an employee has skills in several tools, therefore, there is no functional dependency in this relation. However, this relation has two multivalued attributes: *project* and *tool*.

The attributes *project* and *tool* are assumed to be independent of each other, as any project can use any tool. The following table (not relation) defines this situation:

e# (Employee Number)	Project	Tool
E001	DBMS, Ecommerce	Python, Virtualization
E002	Ecommerce, Bank Automation	PostgreSQL, Virtualization
E003	Bank Automation	PostgreSQL

Figure 6.1: Sample Data

However, to represent this information through 1NF, you need to create the following relation:

e#	project	tool
E001	DBMS	Python
E001	DBMS	Virtualization
E001	Ecommerce	Python
E001	Ecommerce	Virtualization
E002	Ecommerce	PostgreSQL
E002	Ecommerce	Virtualization
E002	Bank Automation	PostgreSQL
E002	Bank Automation	Virtualization
E003	Bank Automation	PostgreSQL

Figure 6.2: 1NF relation of data of Figure 6.1

This relation has no FD and primary key of the relation is composite key (*e#, project, tool*), therefore, the relation is in 3NF and BCNF. However, it suffers from the problem of redundancy, for example, the employee E001 is working on DBMS is appearing twice, so is that E001 knows the tool Python. This may lead to update anomaly. Further, you cannot insert information about a new employee, who knows the tools PostgreSQL, but has not been assigned to any project. In addition, if you remove the employee E003 from the Bank Automation project, the information that E003 has skill in PostgreSQL would be lost. Thus, the relation suffers from update, insertion and deletion anomalies.

How can you now normalise the relation, as this contains no FD? For addressing the issues related to such relations, we may first define the concept of multivalued dependency, which relates to relations that contain more than one multivalued attribute. Let us define the multivalued dependency formally for a relation *R(X, Y, Z)*, where *X*, *Y* and *Z* are a set of attributes.

Multivalued dependency (*MVD*): Given a relation *R(X, Y, Z), a MVD* $X \rightarrow\rightarrow Y$ *will hold in relation R if for a specific value of attribute set X, there is an associated set of values of attribute set Y, such that these*

multiple-associated (zero or more) values of attributes Y depends only on the value of attribute X. Further, values of attribute set Y have no dependence on the value of attribute set Z.

Hence, if *MVD* $X \rightarrow\rightarrow Y$ holds in *R*, another *MVD* $X \rightarrow\rightarrow Z$ would also hold, as the function of the attribute set *Y* and attribute set *Z* is symmetrical.

In the *Example 1* given above, the employee number can determine all the projects, employee is working on, and also an employee number can determine all the tools in which that employee is skillful. Further, both the *project* and *tool* attributes are independent of each other. Therefore, the following MVDs hold in relation of example 1:

$e\# \rightarrow\rightarrow project$
$e\# \rightarrow\rightarrow tool$

Let us now define the concept of MVD in a different way. Consider the relation *R(X, Y, Z)* having a multi-valued set of attributes *Y* associated with a value of *X*. Assume that the attributes *Y* and *Z* are independent, and Z is also multi-valued. Now, more formally, $X \rightarrow\rightarrow Y$ is said to hold for *R(X, Y, Z)* if *t1* and *t2* are two tuples in *R* that have the same values for attributes *X* (*t1[X] = t2[X])* then *R* also contains tuples *t3* and *t4* (not necessarily distinct) such that:

t1[X] = t2[X] = t3[X] = t4[X]
t3[Y] = t1[Y] and *t3[Z] = t2[Z]*
t4[Y] = t2[Y] and *t4[Z] = t1[Z]*

For example, consider Figure 6.2, the two such tuples are:

Tuple 1: E001	DBMS	Python
Tuple 4: E001	Ecommerce	Virtualization

Then two more tuples must exist as follows:

E001	DBMS	Virtualization
E001	Ecommerce	Python

Please check these two are Tuple 2 and Tuple 3 in Figure 6.2

We are, therefore, requiring that every value of *Y* appears with every value of *Z* to keep the relation instances consistent. In other words, the above conditions insist that *Y* and *Z* are determined by *X* alone and there is no relationship between *Y* and *Z*, since *Y* and *Z* appear in every possible pair and hence these pairings present no information and are of no significance. Only if some of these pairings were not present, there would be some significance in the pairings.

(Note: If *Z* is single-valued and functionally dependent on *X* then *Z1 = Z2*. If *Z* is multivalued dependent on *X* then *Z1 <> Z2*).

The theory of multivalued dependencies is very similar to that for functional dependencies. Given a set of MVDs, say *D*, you can find D^+, the closure of *D* using a set of axioms. We do not discuss the axioms here. You may refer to this topic in further readings.

Before explaining the 4NF of a relation, one more term needs to be defined. It is the Trivial MVD, which is defined next.

Trivial MVD: A MVC X $\rightarrow\rightarrow$ Y is called trivial MVD if either Y is a subset of *X* or *X* and *Y* together form the relation R.

The MVD is trivial since it results in no constraints being placed on the relation. For example, consider a relation *dependent* (e#, *edependent#*). An employee can have zero or more dependents, therefore, e# uniquely determines the **values** of *edependent#.* However, this MVD is trivial, as e#, *edependent#* together forms the relation *dependent.* This *dependent* relation cannot be decomposed any further.

Therefore, a relation having non-trivial MVDs must have at least three attributes; two of them would be multivalued and not dependent on each other. Non-trivial MVDs result in the relation having some constraints on it since all possible combinations of the multivalued attributes are then required to be present in the relation. In the next section, we discuss how MVD can be used to decompose a relation into fourth normal form.

6.3 FOURTH NORMAL FORM

In this section, first we define the fourth normal form (4NF) and then present an example about how MVD can be used to decompose a relation to 4NF.

A relation R *is in 4NF if for all the multivalued dependencies* ($X \rightarrow\rightarrow Y$) *any one of the following clauses holds:*

- the multivalued dependencies ($X \rightarrow\rightarrow Y$) is trivial,
- X is a candidate key for R.

The dependency $X \rightarrow\rightarrow \emptyset$ or $X \rightarrow\rightarrow Y$ in a relation $R(X, Y)$ is trivial, since they must hold for all $R(X, Y)$. In this case, $R(X, Y)$ is in 4NF. Similarly, if in a relations $R(A, B, C)$ with only three attributes, if a trivial MVD $(A, B) \rightarrow\rightarrow C$ holds, then $R(A, B, C)$ is in 4NF.

If a relation has more than one multivalued attribute, you should decompose it into the fourth normal form using the following rules of decomposition:

For a relation $R(X,Y,Z)$, if it contains two nontrivial MVDs $X \rightarrow\rightarrow Y$ and $X \rightarrow\rightarrow Z$, then decompose the relation into $R_1(X,Y)$ and $R_2(X,Z)$ or more specifically, if there holds a non-trivial MVD in a relation $R(X,Y,Z)$ of the form $X \rightarrow\rightarrow Y$, such that $X \cap Y = \phi$, that is the set of attributes X and Y are disjoint, then R must be decomposed to $R_1(X,Y)$ and $R_2(X,Z)$, where Z represents all attributes other than those in X and Y.

Intuitively R is in 4NF if

(1) All dependencies are a result of keys.
(2) When multivalued dependencies exist, a relation should not contain two or more independent multivalued attributes.

The decomposition of a relation to achieve 4NF would normally result in not only reduction of redundancies but also avoidance of anomalies.

Example 2: Normalise the relation *emp (e#, project, tool)* shown in Figure 6.2 using the MVDs:

e# $\rightarrow\rightarrow$ project and e# $\rightarrow\rightarrow$ tool.

The relation has no FD, but two MVDs, therefore, the relation is in BCNF but not 4NF. To decompose the relation into 4NF, you may use any of the MVD. The decomposed relation would be:

empproj (e#, project) with MVD e# $\rightarrow\rightarrow$ project, which is now a trivial MVD; and

emptool (e#, tool) with MVD e# $\rightarrow\rightarrow$ tool, which is a trivial MVD.

On decomposition of the relation in Figure 6.2, by taking the projections as stated above, the relational state of the decomposed relations would be:

empproj	
e#	project
E001	DBMS
E001	Ecommerce
E002	Ecommerce
E002	Bank Automation
E003	Bank Automation

emptool	
e#	tool
E001	Python
E001	Virtualization
E002	PostgreSQL
E002	Virtualization
E003	PostgreSQL

Figure 6.3: Decomposition of *emp (e#, project, tool)* relation to 4NF

You can observer the following in Figure 6.3

- The key to relations empproj and emptool are (e#, project) and (e#, tool) respectively.
- There is no redundancy in empproj and emptool relations.
- Both the relations do not suffer from any anomaly.

So far, you are able to decompose a relation based on FD and MVD. Are there any other forms of dependencies? Researcher have shown several kinds of dependencies. However, for this course we will discuss about one more type of dependency, called the join dependency, which is discussed next.

☞ Check Your Progress 1

1) What are Multi-valued Dependencies? When can you say that a constraint X is multi-valued dependency?

……

……

……

2) Identify the MVDs in the following relation. Decompose the relation into 4NF using the MVDs.

EMP

ENAME	PNAME	DNAME
Rohan	Big Data	AI Unit
Rohan	Machine Learning	Analytics
Rohan	Big Data	Analytics
Rohan	Machine Learning	AI Unit

……

……

……

3) Convert the following relation to 4NF relations.

SUPPLY

SNAME	PARTNAME	PROJNAME
XYZ Pvt Ltd	Bolt	Big Data
XYZ Pvt Ltd	Nut	Machine Learning
ABC Ltd	Bolt	Machine Learning
Info Comm Ltd	Nut	AI
ABC Ltd	Nail	Big Data
ABC Ltd	Bolt	Big Data
XYZ Pvt Ltd	Bolt	Machine Learning

……

……

……..

6.4 JOIN DEPENDENCIES

As discussed in the previous section, a relation that suffers from insertion, update and deletion anomalies is decomposed using either FD or MVD. The normal forms require that a given relation *R,* if not in the given normal form, should be decomposed in two relations to meet the requirements of the normal form. However, in some cases, a relation can have problems like redundancy leading to anomalies, yet it cannot be decomposed in two relations without loss of information. In such cases, it may be possible to decompose the relation in three or more relations. When does such a situation arise? Such cases normally happen when a relation has at least three attributes such that all those values are totally independent of each other. It may also be the case when a ternary relationship exists among three entities.

Following example explains this in detail. Consider a relation:

ProjToolEmp (projectid, toolid, empid)

There are no constraints on this relation that is:

- Any project can use any tools
- Any tool can be used by any employee
- Any employee can work on any project
- No employee would use all the tools
- No employee would work on all the projects
- No project uses all the tools
- All three attributes are independent of each other

Assume that the relation has the following relational instance:

Tuple#	projectid	toolid	empid
1	P1	T1	E1
2	P2	T2	E2
3	P1	T2	E2

Figure 6.4: A ternary relation with all independent attributes

The relation above does not have any FDs and MVDs since the attributes projectid, toolid and empid are independent; they are related to each other only by the pairings that have significant information in them.

For example, the first tuple of relation states that employee E1 works on P1 project, which uses tool T1. The key to the relation is the composite key (projectid, toolid, empid). The relation is in 4NF, but still suffers from the insertion, deletion, and update anomalies, as the information that Employee E1 can use tool T1 is redundant in tuple 3. Similarly, information like project P3 uses tool T1 cannot be inserted in the relation, as no employee has started working on the project. Likewise, on deleting tuple 2 from the relation, may delete the information like Employee E2 can use tool T2. Therefore, you need to decompose the relation to minimise the anomalies. As there are no FDs or MVDs in this relation, there is no basis of decomposition. You may try to see what happens when you decompose the relation into the following two relations:

ProjectTool

projectid	toolid
P1	T1
P2	T2
P1	T2

ProjectEmployee

projectid	empid
P1	E1
P2	E2
P1	E2

Figure 6.5: A decomposition of ternary relation of Figure 6.4

What happens when you take Join of the two relations on the attribute *projectid:*

Tuple#	projectid	toolid	empid
1	P1	T1	E1
2	*P1*	*T1*	*E2*
3	P2	T2	E2
4	*P1*	*T2*	*E1*
5	P1	T2	E2

Figure 6.6: *ProjectTool* JOIN *ProjectEmployee*

You may notice that the tuples 2 and tuple 4 in the relation state in Figure 6.6 were not existent in original relational state given in Figure 6.4. Thus, this decomposition is a lossy decomposition. So, what should be done to remove the anomalies?

In such situation, you may have to decompose the relation into three relations. Two of which are already shown in Figure 6.5. A third relation as shown in Figure 6.7 must be added.

ToolEmployee

toolid	empid
T1	E1
T2	E2

Figure 6.7: The third relation

In order to obtain the original relation, you need to perform the following join operation:

ProjectTool JOIN *ProjectEmployee* JOIN *ToolEmployee*

Figure 6.6 represents (*ProjectTool* JOIN *ProjectEmployee*) which can be joined with *ToolEmployee*

Tuple#	projectid	toolid	empid
1	P1	T1	E1
~~*2*~~	~~*P1*~~	~~*T1*~~	~~*E2*~~
2	P2	T2	E2

~~4~~	~~P1~~	~~T2~~	~~E1~~
3	P1	T2	E2

Figure 6.8: *ProjectTool* JOIN *ProjectEmployee* JOIN *ToolEmployee*

Please note that Tuple 2 and Tuple 4 of Figure 6.6 will not be part of Figure 6.8, as there are no matching tuples in Figure 6.7. Thus, you can observe that Figure 6.4 and Figure 6.8 are identical.

Therefore, the relation ProjToolEmp (projectid, toolid, empid), which has no FD and MVD can be decomposed into three relations, which can be joined to form the original relation. This forms the concept of join dependency, which is explained next.

Join Dependency (JD): A relation R satisfies join dependency over the projections ($P_1, P_2, ..., P_n$) if and only if every instance of *R*, the join of $P_1, P_2, ..., P_n$ creates the instance of *R.*

For example, the instance of ternary relation ProjToolEmp (projectid, toolid, empid), as shown in Figure 6.4, when decomposed to two relations, as shown in Figure 6.5, does not satisfy the join dependency. This is shown in Figure 6.6. On addition of the third relation, as shown in Figure 6.7, the three projections ProjectTool (projectid, toolid), ProjectEmployee(projectid, empid) and ToolEmployee (toolid, empid) satisfies the join dependency as JOIN on the three projections, viz. ProjectTool (projectid, toolid), ProjectEmployee(projectid, empid) and ToolEmployee (toolid, empid), is the same as the instance of the relation. Thus, the three projections, as stated above, satisfies the join dependency.

6.4 FIFTH NORMAL FORM

The fifth normal form (5NF) (Project-Join normal form (PJNF)) deals with join-dependencies, which is a generalisation of the MVD. The aim of the fifth normal form is to have relations that cannot be decomposed further. A relation in 5NF cannot be constructed from several smaller relations.

A relation R is in 5NF if for all the join dependencies any one of the following clauses holds:

(a) join-dependency ($P_1, P_2, ..., P_n$) is trivial (that is, one of the P_i's is R)

(b) Every P_i is a super key of R.

For example, the instance of ternary relation ProjToolEmp (projectid, toolid, empid), as shown in Figure 6.4, has a Join Dependency (ProjectTool (projectid, toolid), ProjectEmployee(projectid, empid), ToolEmployee (toolid, empid)). Therefore, this relation is not in 5NF, as it violates the clauses of the 5NF, given above. You can decompose the relation ProjToolEmp (projectid, toolid, empid) into its projections as follows:

ProjectTool (projectid, toolid)

ProjectEmployee(projectid, empid)

ToolEmployee(toolid, empid)

Each of these three relations are in 5NF, as they have trivial join dependency. The instance of each of the decomposed relations is shown in Figure 6.5 and Figure 6.7. You may also observe that this decomposition is lossless join decomposition. It may be noted that every *MVD* is also a join dependency. Therefore, every *PJNF* (*5NF*) schema is also in *4NF*. A relation schema that has Join Dependencies and suffers from anomalies, can be decomposed into projections, as per the join dependency. The new relations would be in *PJNF* schema.

☞ **Check Your Progress 2**

1) What is join dependency?

……

……..

2) Define 5NF.

……

……

……..

3) Convert the relational instance of SUPPLY relation, as given in question 3 of check your progress 1. to 5NF.

……

……

……..

6.5 OTHER NORMAL FORMS

The researchers of database systems have found additional dependencies and normal forms. This section introduces the basic concept behind these forms. First, we define some additional types of dependencies.

Inclusion Dependency

The inclusion dependency has been designed for two specific types of database constraints that are not defined by the concept of FD, MVD and Join dependency. These two constraints are:

- Foreign key constraints
- Class/subclass relationships

Please note that these constraints are between two relations. Therefore, it requires a new form of formal definition. We define it in the context of foreign key constraints.

Consider two relations R1 and R2, which are related through a foreign key constraint such that a set of attributes in R1, say X, is the foreign key that references the relation R2 on a set of attributes, say Y. Please note that attribute sets X and Y must have similar number of attributes and similar domains, so that foreign key constraint is applicable. The inclusion dependency for such a situation will be defined as follows:

An **inclusion dependency (**denoted as $R1.X < R2.Y$) if the following relationship between the projections holds:

$$\pi_X(r1) \subseteq \pi_Y(r2) \quad (1)$$

Where r1 and r2 are the instances of relation R1 and R2 respectively at the same instance of time.

For example, consider the following two relational instances:

Relation: DEPT

deptID	deptName	deptlocation
D01	Marketing	Delhi
D02	Production	Mumbai
D03	HR	Delhi

Relation: EMP

empID	empName	salary	department
E01	ABC	30000	D01
E02	BCD	40000	D01
E03	CDE	45000	D02
E04	EFG	35000	D02

There exists a foreign key between the two relations, viz. department in EMP relation references deptID in DEPT relation. Therefore, the inclusion dependency *EMP.department* < *DEPT.deptID* must hold for the given relational instances.

The equation (1) for this may be (assuming that relational instance of DEPT is dept and EMP is emp):

$\pi_X(r1)$ is $\pi_{department}(emp)$, which would be:

emp

department
D01
D02

and $\pi_Y(r2)$ is $\pi_{deptID}(dept)$, which would be:

dept

deptID
D01
D02
D03

You may observe that the inclusion dependency *EMP.department* < *DEPT.deptID* holds, as $\pi_{department}(emp) \subseteq \pi_{deptID}(dept)$.

Template Dependency

The template dependency is specified in the form of a template and can be used to represent any generic dependency. These dependencies can define any constraints in the form of a template. A template consists of two parts – the first part which shows the tuples that may exist in a relation is called the hypotheses, which are followed by the conclusion of the template.

A template dependency can be used to represent any dependency in this form. The following example shows how a FD can be represented using a template dependency.

Example: Consider a relation RESULT (studentid, courseid, grade) with the FD

studentid, coursed→grade

Represent this FD using template dependency.

Description	studentid	courseid	grade
Hypothesis	S1	C1	G1

	S1	C1	G2
Conclusion	G1 = G2		

The following example shows, how MVDs can be represented using template dependency.

Example: Consider a relation EMPLOYEE (e#, project, tool) with the set of MVDs

e# →→ project and e# →→ tool.

The following template dependency defines these MVDs:

Description	e#	project	tool
Hypothesis	E1	P1	T1
	E1	P2	T2
Conclusion	E1	P1	T2
	E1	P2	T1

One example of this MVD is shown with attribute values in the following table:

Description	e#	project	tool
Hypothesis	E001	DBMS	Python
	E001	Ecommerce	Virtualization
Conclusion	E001	DBMS	Virtualization
	E001	Ecommerce	Python

However, the actual use of template dependency may be to represent the constraints that cannot be represented using FD, MVD and join dependency. The following example explains this.

Example: Consider the relation EMP (empID, empName, salary, headID). In this relation the attribute headID represents empID of the head of the employee. You want to put a constraint in the relation that the employee cannot be given more salary than his/her head. The following template dependency would define this constraint:

Description	empID	empName	salary	headID
Hypothesis	E1	N1	S1	E2
	E2	N2	S2	E3
Conclusion	S2 >= S1			

The Domain-Key Normal Form (DKNF)

The Domain-Key Normal Form (DKNF) is beyond 5NF and proposes to make a set of relations free of all anomalies. The DKNF was defined as a consequence of Fagin's theorem that states:

"A relation is in DKNF if every constraint on the relation is a logical consequence of the definitions of keys and domains."

Let us define the key terms used in the definition above – *constraint, key* and *domain* in more detail. These terms were defined as follows:

Keys can be either the primary keys or the candidate key. Let R be a relation schema with $CK \subseteq R$. A key CK requires that CK be a super key for schema R such that $CK \rightarrow R$. Please note that a key declaration is a functional dependency but not all functional dependencies are key declarations.

Domain is the set of definitions of the contents of attributes and any limitations on the kind of data to be stored in the attribute. Let A be an attribute and **dom** be a set of values. The domain declaration can be stated as $A \subseteq$ **dom**. It requires that the values of A in all the tuples of R be values from the set **dom**.

Constraint is a well-defined rule that is to be upheld by any set of legal data of R. A *general constraint* is defined as a predicate on the set of all the relations of a given schema. The MVDs, JDs are the examples of general constraints. However, a general constraint need not be just functional, multivalued, or join dependency. For example, the first two digits of your enrolment number represent the year in which you have taken admission. Assuming that MCA is the only programme of the university, whose maximum duration is 4 years. Also, assume that admission to this programme was started in 2021. Therefore, in the year 2026, there would be two types of students, viz. students whose enrolment number starts with 21 and students whose registration number starts with 22, 23, 24, 25 and 26. The registration of the students, whose registration starts with 21 would become invalid. Therefore, the general constraint for such a case may be:
"If the first two digits of *t*[*enrolmentnumber*] is 21, then *t*[*marks*] are valid."
The *t* represents a tuple and *enrolmentnumber* and *marks* are the attributes.

The constraint suggests that the database design is not in DKNF. To convert this design to DKNF design, you need two schemas as:

Validstudentschema =(enrolmentnumber, subject, marks)

Invalidstudentschema = (enrolmentnumber, subject, marks)

Please note that the schema of valid students requires that the enrolment number of the students begin with 22. The resulting design is in DKNF.

Although DKNF is an aim of a database designer, it may not be implemented in a practical design.

☞ Check Your Progress 3

1) Define Inclusion Dependencies.

……

……

……

2) Define the template dependency.

……
……

3) What is the key idea behind DKNF?

……
……
……

6.6 SUMMARY

This unit explains the concept of multi-valued dependencies, which causes a relation to have data redundancy. This causes a relation to have data anomalies. MVD is a consequence of having a set of attributes in a relation that determines more than one multi-valued attribute, which are independent of each other. MVD forms the basis of decomposition of a relation into 4NF relations. Further, certain relations do not have any FDs and MVDs but have anomalies. Such relations, in general, consist of more than two independent attributes. These relations contain join dependency, that is the relation has several projections, which can be joined losslessly to produce original relation. The join dependency forms the basis for 5NF decomposition. The unit also discusses about the inclusion and template dependencies, which are designed to represent the constraints that cannot be assigned using FDs, MVDs and join dependencies. Finally, the unit introduces the concept of DKNF. You may refer to the further readings for more details on these topics.

6.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) An MVD is a constraint due to multi-valued attributes. A constraint is multi-valued if all the following conditions hold in a relation:
   - The relational must have at least three attributes out of which one should be multi-valued attribute.
   - One of the attributes can multi-determine the other two attributes.
   - The other two attributes, as stated above should be independent of each other.

2) There are two MVDs that exist in the relation. These are:

ENAME $\rightarrow\rightarrow$PNAME and ENAME $\rightarrow\rightarrow$DNAME

Due to these two MVDs the relation suffers from insertion, update and deletion anomalies. This relation can be decomposed into the following projections, which are in 4NF.

EMP_PROJECT

ENAME	PNAME
Rohan	Big Data
Rohan	Machine Learning

EMP_DNAME

ENAME	DNAME
Rohan	AI Unit
Rohan	Analytics

3) The relational instance of SUPPLY relation shows that all three attributes are independent of each other as any supplier can supply any part to any project. Thus, there is no FD or MVD in the relation. Therefore, the relation is already in 4NF.

Check Your Progress 2

1) A join dependency is defined for a relation *R* and its projections, say *R1, R2, ...Rn*, as follows: The join of relations *R1, R2, ...Rn* must be equal to the relation *R*.
2) A relation is said to be in 5NF if either it has trivial join dependencies, or every projection *Ri* of the relation *R* is a super key of *R*.
3) All the attributes of relation SUPPLY are independent of each other. Does the relation have join dependency (SNAME, PARTNAME), (PARTNAME, PROJNAME), (PROJNAME, SNAME)? The three projections for the given instance of SUPPLY would be:

R1

SNAME	PARTNAME
XYZ Pvt Ltd	Bolt
XYZ Pvt Ltd	Nut
ABC Ltd	Bolt
Info Comm Ltd	Nut
ABC Ltd	Nail

R2

PARTNAME	PROJNAME
Bolt	Big Data

Nut	Machine Learning
Bolt	Machine Learning
Nut	AI
Nail	Big Data

R3

SNAME	PROJNAME
XYZ Pvt Ltd	Big Data
XYZ Pvt Ltd	Machine Learning
ABC Ltd	Machine Learning
Info Comm Ltd	AI
ABC Ltd	Big Data

The join of the first two projections (R1 and R2) on PARTNAME would be:

JoinR1R2

SNAME	PARTNAME	PROJNAME
XYZ Pvt Ltd	Bolt	Big Data
XYZ Pvt Ltd	Bolt	Machine Learning
XYZ Pvt Ltd	Nut	Machine Learning
XYZ Pvt Ltd	Nut	AI
ABC Ltd	Bolt	Big Data
ABC Ltd	Bolt	Machine Learning
Info Comm Ltd	Nut	Machine Learning
Info Comm Ltd	Nut	AI
ABC Ltd	Nail	Big Data

The join of relations JoinR1R2 with R3 on SNAME, PROJNAME would be:

SNAME	PARTNAME	PROJNAME
XYZ Pvt Ltd	Bolt	Big Data
XYZ Pvt Ltd	Bolt	Machine Learning
XYZ Pvt Ltd	Nut	Machine Learning
ABC Ltd	Bolt	Big Data
ABC Ltd	Bolt	Machine Learning
Info Comm Ltd	Nut	AI
ABC Ltd	Nail	Big Data

Please observe that the joined relation is same as the instance of relation SUPPLY. Therefore, the join dependency (SNAME, PARTNAME), (PARTNAME, PROJNAME), (PROJNAME, SNAME) holds over the relation SUPPLY. To convert SUPPLY relation to 5NF, you may decompose it into the three projections R1, R2 and R3 as shown above. Please notice that each of the relation R1, R2 and R3 now have trivial join dependency, therefore, are in 5NF.

Check Your Progress 3

1) An inclusion dependency defines the referential constraint on two attributes. An inclusion dependency holds if a projection on an attribute of a relation, which may be a foreign key, is a proper subset of projection of another attribute of another relation, where it is the primary key.

2) Template dependency is a generic representation of various dependencies. It consists of two parts – the hypothesis and conclusion.

3) DKNF is the "ultimate normal form". It defines the terms keys, domains and constraints.

UNIT 7 STRUCTURED QUERY LANGUAGE

7.0 INTRODUCTION

As discussed in the previous Units, a relational database system consists of relations, which are normalised to minimise redundancy. The normalisation process involves the concepts of functional dependency (FD), multi-valued dependency (MVD) and join dependency (JD). The normalisation results in the decomposition of tables into smaller but non-redundant relations. After designing the normalised database system, you would like to implement it by using software called relational database management system (RDBMS). RDBMSs were designed to manage relational data. Some of these RDBMSs are – Oracle, DB2, SQL Server, MySQL, etc. All these RDBMSs are designed and developed by different organisations and use different data storage formats. A structured query language (SQL) is one of the standards, which was created for the transfer of information from a RDBMS. The SQL consists of three basic languages, viz. Data Definition Language (DDL), which is used for defining the structure of the relations in the form of SQL tables and indexes; Data Manipulation Language, which is used for input, modification and deletion of information in SQL tables; and Data Control Language, which is used for controlling the access rights on the data of SQL tables and indexes. This unit introduces these three languages and the basic set of commands used in SQL.

You must learn SQL to use database technologies, therefore, you are advised to go through this unit very carefully. You must practice the concepts learnt in this unit on a commercial RDBMS.

7.1 OBJECTIVES

After going through this unit, you should be able to:

- create SQL objects (tables, indexes, etc.) from a database schema;
- insert data into database tables using SQL commands;
- retrieve data from a database using SQL queries;
- use the Group By and Having clauses of SQL;
- Using aggregate functions of SQL;
- Create access rights on different database objects using SQL.

7.2 INTRODUCTION TO SQL

The Structured Query Language (SQL) was initially designed at IBM by D D Chamberlin and R F Boyce for the relational model that was proposed in 1970 by E F Codd. SQL was designed as a fourth-generation programming language to produce multiple record output as a result of a single command. In addition, SQL commands did not require the specification of indexes or other information for data retrieval. Later, SQL was standardised by the American National Standard Institute (ANSI) and a number of standards of SQL have been created, starting from SQL:86, which was proposed in 1986, SQL:92, SQL:99 and SQL:2003. Several modifications have been proposed in SQL:2003 in the years 2006-2019. One of the key advantages of using SQL is that it is *the query language* of most RDBMSs. Thus, it facilitates the migration of a database from one RDBMS to another RDBMS. However, some differences can be found in the SQL implementation on several RDBMSs. This unit covers some of the basic features of SQL only.

Why did SQL become popular? One of the major reasons for SQL's popularity is that it allows you to write queries without specifying how the query would be executed. For example, in case you want to JOIN three relations or tables, say A, B, C, then SQL allows you to join these tables without specifying the sequence of joining. The decision about how to execute the joins, e.g. (A JOIN B) JOIN C or A JOIN (B JOIN C), what indexes and views are to be used, etc.; are left to the query parser, translator, and query optimiser of RDBMS. In addition, the syntax of SQL is closer to the English language. Thus, SQL queries are simpler to write and run. The following are the features of SQL in the context of RDBMS:

- SQL is non-procedural, as in SQL you just need to say what information is required by you and NOT how that information is to be acquired from the database.
- The syntax of SQL is closer to the English Language; thus, it is very easy to comprehend.
- In general, the output of a SQL query may be a single record or a group of records.
- An interesting feature of SQL is that it can be used at different levels of ANSI's three-level architecture of database system.

SQL includes the following three sub-languages:

- Data Definition Language (DDL): The basic focus of DDL is the commands that can be used to convert database schema into SQL tables, indexes, etc., or change of structure of the tables. These commands also allow you to define the constraints on the attributes of a table. The DDL commands are explained in the next section.
- Data manipulation language (DML): The purpose of data manipulation is twofold. First, it has commands to input, modify and delete the data in the SQL tables. Second, it has the SELECT command, which helps in the retrieval of data from one or multiple tables.
- Data control language (DCL): The purpose of these commands is to specify user's privileges to database users. These commands are very important in client/server databases with different types of users.

7.3 DATA DEFINITION LANGUAGE

The design of a database consists of the physical, conceptual, and external schema. To implement this design, you need to define these schemas using SQL. In this unit, we use commands to define the conceptual schema. The external schema-related commands are discussed in the next unit. As far as the physical schema is concerned, we will cover only a few commands. The conceptual schema, which

includes the relational schema and attributes, is implemented in RDBMS as a set of tables and columns, whereas the tuples of the relations are implemented with the help of rows or data records.

We will explain some of the basic DDL commands in this section with the help of an example. Consider two relations of a UNIVERSITY database system, namely STUDENT and PROGRAMME.

The student relation (STUDENT) has the following data: student ID (STID), which is also the primary key, the name of the student (STNAME), the programme code in which that student is registered (PROGCODE) and the mobile phone number of the student (STMOBILE). Please note that this schema assumes that a student is allowed to register in only a single programme for which s/he is allotted a unique student ID.

The second relation, PROGRAMME, has the following data: programme code (PROGCODE), which is also the primary key, the name of the programme (PROGNAME) and the fee for that programme.

STUDENT (STID, STNAME, PROGCODE, STMOBILE) and
PROGRAMME (PROGCODE, PROGNAME, FEE)

The first step would be to define the datatypes of various attributes. We propose to use the following data types for the attributes.

	Relation Name	STUDENT		
Attribute	STID	STNAME	PROGCODE	STMOBILE
Data type	Character	Character	Character	Number
Length	4	40	6	12 digits
Constraint	PRIMARY KEY	NOT NULL	FOREIGN KEY References PROGRAMME table	-

	Relation Name	PROGRAMME	
Attribute	PROGCODE	PROGNAME	FEE
Data type	Character	Character	DECIMAL
Length	6	30	5
Constraint	PRIMARY KEY	NOT NULL	>1000 and < 50000

Figure 1: Example Relations

Once you have completed the data design, the next step would be to use SQL to create the SQL tables. To do so, you should learn about different data types in SQL, commands for creating tables in SQL, commands for creating constraints, etc. Let us discuss each of these.

Data Types in SQL: SQL supports many data types. Figure 2 lists some of the commonly used data types of SQL. For more data types supported by a DBMS, you may refer to the information manual of that DBMS. You may select one of these data types for each column.

CHAR (*n*)	It accepts a character string of size *n*. The character string can include alphabets, numeric digits, and special characters. The size of each string is fixed, that is, *n*.

VARCHAR (*n*)	It accepts a character string **up to** the size *n*. The character string can include alphabets, numeric digits, and special characters.
BOOLEAN	It accepts a value False (zero value) or True (non-zero value)
INTEGER (*n*)	It can store signed integers or unsigned integers of length 32 bits. *n* represents the display width of the integer.
DECIMAL (*n, d*)	It can store decimal numbers of size *n* having *d* digits after the decimal point. The default value of *d* is 0.
DATE	Represents a valid date. It uses 4 digits for years and 2 digits each for month and day.
TIMESTAMP	It assigns a timestamp, which can be used to determine the recency of data.

Figure 2: Some Basic Data Types

For the relations of Figure 1, you may select CHAR for STID, PROGCODE, STMOBILE (as it is not required for computation), and VARCHAR for STNAME and PROGNAME, as the names of students may be of different lengths; so are the names of the programme's. The data type of FEE is DECIMAL(5).

Creating the Database and the Tables: In order to create the tables as given in Figure 1, you need to first create a database using the following command:

```
CREATE DATABASE <name of the database>;
USE <name of the database>; //This command is needed when the DBMS has multiple databases.
```

The following commands will create the UNIVERSITY database:

```
CREATE DATABASE UNIVERSITY;
USE UNIVERSITY;
```

Now, you can create the tables using the create table command. The following is the syntax of this command:

```
CREATE TABLE  <name of the table>  (
        ColumnName1 <data type> [constraints],
        ColumnName2 <data type> [constraints],
        …
        );
```

The following are the descriptions of the create table command:

- The name of a table should begin with an alphabet. It should not contain blank space and special characters except the underscore character (_).
- You should not use reserved words as a name of a table.
- You should use a unique name for each column in a table.
- The constraints are optional and therefore, shown in [ ] brackets.
- The data type should include the size of the column.
- You may use any of the following constraints on the column:

NOT NULL	This column of a table cannot be left blank while data entry or modification.
UNIQUE	This value should be unique across all the values in this column.
PRIMARY KEY	The column is or is a part of the primary key.
CHECK	It is followed by certain conditions, which should be fulfilled by the column.
DEFAULT	When a default value is specified for a column.
REFERENCES	This is used for specifying referential constrains, while creating a table.

Figure 3: Some of the constraints in Create Table Command

Now, you are ready to create the tables. Since the table STUDENT contains a reference to PROGRAMME table, therefore, you may create the PROGRAMME table first using the following command:

```
CREATE TABLE PROGRAMME
    (
        PROGCODE CHAR(6) PRIMARY KEY,
        PROGNAME VARCHAR(30) NOT NULL,
        FEE DECIMAL (5),
            CHECK (FEE>1000 AND FEE<50000)
    );
```

Now, you are ready to create the STUDENT table. You will use the following command to create the table.

```
CREATE TABLE STUDENT
    (
        STID CHAR(4) PRIMARY KEY,
        STNAME VARCHAR(40) NOT NULL,
        PROGCODE CHAR (6),
        STMOBILE CHAR (12),
        FOREIGN KEY PROGCODE REFERENCES PROGRAMME (PROGCODE)
            ON DELETE RESTRICT
    );
```

The Referential Action: Please note that in the command of creating the STUDENT table, we have used a referential action RESTRICT, which will make sure that any deletion of a record in PROGRAMME table will be restricted in case even one record of STUDENT table contains that PROGCODE. Thus, this action will ensure that there is no violation of the referential integrity constraint during deletion of a record from the PROGRAMME table. The other possible referential action is CASCADE. In this case, the deletion of a record in PROGRAMME table will result in the deletion of all the records of all the students whose PROGCODE is the same as the record, which is getting deleted in the PROGRAMME table.

Creating an Index: You may notice that the primary key of the STUDENT table is STID, therefore, the STUDENT table would be organised in the order of STID. However, many database queries may require the student records in the order of PROGCODE. This would require you to create an index on PROGCODE in the STUDENT table to enhance the query performance. The following command can be used to create an index:

```
CREATE [UNIQUE] INDEX <name of the index>
ON <name of the table> (ColumnName1, ColumnName2, …)
```

You use the keyword UNIQUE, only if a unique index is to be created. For the given example, you may create the following index for the STUDENT table.

```
CREATE INDEX PROGINDEX ON STUDENT (PROGCODE);
```

Other DDL commands: There are a large number of DDL commands. We will discuss only a few commands here. You may refer to the DBMS documentation for more commands.

Commands to alter a Table: For altering a table, you may use an ALTER TABLE command. This command can be used for performing the following functions:

- You can add a new column to an existing table, or you can modify a column of an existing table, by using the command:
  ALTER TABLE <name of the table> ADD/MODIFY (ColumnName1, <datatype>, …);
- You can add a new constraint in a table using the following command:
  ALTER TABLE <name of the table> ADD CONSTRAINT
  <name of the constraint> <type of the constraint> (ColumnName);
- You can drop a constraint on a table or enable or disable it using the following command:
  ALTER TABLE <name of the table> DROP/ENABLE/DISABLE <name of the constraint>;

Commands to delete a Table or an Index: You can remove a table or an index, which is not required any more. The following commands can be used for these purposes.

DROP TABLE<name of the table>;
DROP INDEX <name of the Index> ON <name of the table>;

Commands to create a Domain: As defined earlier, SQL has a large set of data types. However, in many database implementations, you need to create a more meaningful domain that can define the data types and constraints on the data of a specific attribute or column. The following command can be used to create domains. It may be noted that the following command may not be defined in many DBMSs.

CREATE DOMAIN <Name of the Domain> AS <Data type> CHECK (<Constraints>);

You may refer to the DBMS documentation for more details.

Check Your Progress 1

1) List the advantages and disadvantages of using SQL.

……

……

……

……

2) Consider a hotel that has three tables: ROOM (RNo, RType, RRent,), BOOKING (CustID, RNo, BookedFrom, BookedTo) and CUSTOMER (CustID, CustName, CustAddress, CustPhone). Assume the structure of these tables and create the tables using SQL. You may assume the following constraints for the HOTEL database:

- The type of hotel rooms can be: Single Room and Double Room, default room type should be Single room.
- The room rent is between 5000 per day to 25000 per day.
- Rooms are on different floors and numbered from 101 to 150.
- While booking the room the BookedFrom and BookedTo should follow the following relationship: Today's Date <= BookedFrom <= BookedTo
- No room should be booked twice for a specific date.

……..……

……..……

……..……

……..…….

……..

7.4 DATA MANIPULATION LANGUAGE

Once you have created a database and database tables along with the necessary constraints, the next step is to insert data in the tables. While inserting the data in the tables, you may commit some mistakes or there may be the need of making certain changes in the data of the tables, therefore, you would be requiring SQL commands to INSERT, UPDATE and DELETE records in a database table. These are Data manipulation language (DML) commands. These DML commands allow you to input and edit the data in the tables. Further, you would like to retrieve selected information from the database. In this section, first, we discuss the command to insert, update or delete the data followed by commands to retrieve information from the database. You may please note that the changes made by the DML statements are made permanent only after these operations are COMMITTED. You will learn about COMMIT in Unit 9.

7.4.1 Data Insertion, Updating and Deletion

The DML commands are used for inputting and editing data in a database table. To insert data in a table, you may use the insert command, which is explained next.

Inserting Data: The following command is used to insert a record into a table. The following two formats of insert commands are used:

If you are inserting a record that had data for all the columns, then you can simply use the command format:

INSERT INTO <name of the table> VALUES (v1, v2, v3, …);

Please note that v1, v2, etc. are the values that are to be inserted into the respective column of the database. For example, to insert data into the PROGRAMME table, you can use the following INSERT command:

```
INSERT INTO PROGRAMME
        VALUES ("PGDCA", "Postgraduate Diploma in Computer Applications", 22000);
```

Please note the following with respect to the insert command, given above:

- The values inserted into the table would be PROGCODE as *PGDCA*; PROGNAME as *Postgraduate Diploma in Computer Applications*; and FEE as *22000*.
- You can use parameters instead of actual values, for example, you can use INSERT INTO PROGRAMME VALUES (&1, &2, &3); These parameter values can be input at the time of execution of the query.
- You can use a sub-query (which will be explained in the next unit) instead of the values given in the command in the (…).

However, if you do not want to insert values in all the columns of the table, then you need to use the following format:

INSERT INTO <name of the table> (C1, C2, C3, …, Cn) VALUES (v1, v2, v3, …, vn);

Here, C1, C2, … represents the column names and v1, v2, … represents the corresponding value of a column. Please note that you need to specify *n* values for the *n* columns. For example, suppose you do not know the mobile number of a new student, still his/her information can be entered in the STUDENT table using the following SQL command:

```
INSERT INTO STUDENT (STID, STNAME, PROGCODE)
            VALUES ("1001", "RAMESH SHARMA", "PGDCA");
```

Please note the following with respect to the insert command, given above:

- The insertion is possible as STMOBILE has no constraint. It can be left blank.
- The values inserted into the table would be STID as *1001*; STNAME as *RAMESH SHARMA*; PROGCODE as *PGDCA*; and STMOBILE as NULL.
- Please also note that the above insert statement will not cause a violation of foreign key constraint, as we have already inserted the PGDCA programme details in the PROGRAMME table.

Updating Data: For updating data, you may use the update command of SQL. The format of the command is given below:

```
UPDATE <name of the table>
SET <C1> = <v1>
WHERE <conditional statement>;
```

For example, to update the mobile number data of student 1001 to the number, say 8484848484, you can use the command:

```
UPDATE STUDENT
SET STMOBILE = "8484848484"
WHERE STID = "1001";
```

You can also use a subquery in an update statement (subqueries are covered in unit 8). For example, consider the fee of all the programmes, which has more than 100 students, is raised by 5%, then the following update command may be used to update the PROGRAMME table.

```
UPDATE PROGRAMME
SET FEE = FEE * 1.05
WHERE PROGRAMME.PROGCODE IN (   SELECT STUDENT.PROGCODE
                                FROM STUDENT
                                GROUP BY STUDENT.PROGCODE
                                HAVING COUNT(*) > 100);
```

The purpose of this subquery will be clear after you go through the next subsection. The subquery will find those programmes which have more than 100 students.

Deleting Data: You can use the following command to delete one or more records from a table:

```
DELETE FROM <name of the table>
WHERE <conditional statement>;
```

The delete command may delete one or more data records at a time. For example, you can try deleting the data of the PGDCA programme from your database, using the following command.

```
DELETE FROM PROGRAMME
WHERE PROGCODE= "PGDCA";
```

However, as there exists a foreign key constraint with referential action RESTRICT and you have already inserted a student who has PGDCA as his programme, therefore, DBMS will not allow you to delete the programme PGDCA from the PROGRAMME table. In case, you want to do so you need to first delete records of all the students of PGDCA from the STUDENT table and then you can delete the PGDCA programme from the PROGRAMME table. Please note that you can use subqueries instead of conditional statement in deletion also.

7.4.2 Data Retrieval

One of the most popular features of any DBMS is the ad-hoc query facility, which requires data retrieval as per the need and access rights of the user. SELECT statement is one of the most used statements of DML, as it helps in the retrieval of requisite data. In this section, we discuss various clauses of this statement.

SELECT Statement: The following is the basic format of the select statement**.**

```
SELECT    <List of Column names or expressions to be displayed>
FROM      <List of Tables that contain the data, whose columns are in select>
WHERE     <Conditions for selection of records for display> ;
```

The following are examples of the use of this statement for the retrieval of data from the two relations given in Figure 1.

Example 1: List the details of all the programmes of the University.

For answering this query, are using one wildcard character (*), which represents all the columns of a table. Please note that in case, you have used * in an arithmetic expression in the SELECT clause, it will be treated as a multiplication sign.

```
SELECT    *
FROM      PROGRAMME;
```

This statement will display the PROGCODE, PROGNAME and FEE of all the records in the PROGRAMME table.

Example 2: List the programme code and programme names of all the programmes of the university, whose fee is less than or equal to 5000.

```
SELECT    PROGCODE, PROGNAME
FROM      PROGRAMME
WHERE     FEE <= 5000 ;
```

Please note the select clause now contains only those column names, which are to be displayed.

Example 3: List the id, name and mobile number of all the PGDCA students.

```
SELECT    STID, STNAME, STMOBILE
FROM      STUDENT
WHERE     PROGCODE = “PGDCA” ;
```

Example 4: This example shows the use of the **arithmetic operator** and **alias**. What would be the fee of the PGDCA programme, if students are given a discount of 15%?

```
SELECT    PROGCODE, PROGNAME, FEE*0.85 AS DISCOUNTEDFEE
FROM      PROGRAMME
WHERE     PROGCODE = “PGDCA” ;
```

Please note the computation of the fee and assigning the result of the computation a new name, that is an alias, DISCOUNTEDFEE. Please note that the syntax of assigning an alias may be different in different RDBMS.
Even in SQL expressions, the operator precedence is followed. Further, these expressions are executed from the left towards the right.

Example 5: This example shows the use of the **concatenation operator**. If you want to display the names of the programme as: "PGDCA: Postgraduate Diploma in Computer Applications" programme, you should use the following SQL command.

```
SELECT    PROGCODE + ":" + PROGNAME
FROM      PROGRAMME
WHERE     PROGCODE = "PGDCA" ;
```

Please note that in this statement the concatenate operator is '+'. However, in different DBMSs it may be different, for example, MySQL uses CONCAT( ) function, whereas Oracle uses || as a concatenation operator. In addition, please also note the use of the literal character string ":" in the command. A literal string may not be part of any column value but can be added to enhance information display.

Example 6: This example demonstrates how duplicate rows can be eliminated using the **DISTINCT** operator. Find the programme code of those programmes, which has at least one student.
To find the answer to this query, you may use the STUDENT table and Project it on programme code.

```
SELECT    DISTINCT PROGCODE
FROM      STUDENT
```

In case you do not use DISTINCT then you will get duplicate values of programme code.

Example 7: This example demonstrates the use of the range operator **BETWEEN … AND**. To find the list of programmes whose fee is >=5000 but <= 15000, you may use the following command:

```
SELECT    *
FROM      PROGRAMME
WHERE     FEE BETWEEN 5000 AND 15000;
```

Please note that both the values, viz. 5000 and 15000 are included in the range.

Example 8: This example demonstrates the use of the set operator **IN**. Find the students of PGDCA or BCA programmes. One way of answering that query would be:

```
SELECT    *
FROM      STUDENT
WHERE     PROGCODE IN ("BCA", "PGDCA") ;
```

Example 9: This example demonstrates the use of **LIKE** operator for matching a pattern of characters in the columns which are of CHAR or VARCHAR type. It may be noted that LIKE operator is supported by several wildcard characters, which may be different in different DBMS. In this example, we demonstrate the use of % wildcard character that matches zero or many characters in a string. For example, a string like %COM% will match with strings: **COM**PUTER, **COM**MERCE, IN**COM**E, M**COM** etc.

To find the list of all the programmes, which have word "Computer", you may give the command:

```
SELECT    *
FROM      PROGRAMME
WHERE     PROGNAME LIKE "%Computer%";
```

Example 10: This example demonstrates the use of **IS NULL** operator. Find the list of students, whose mobile number is not with the University.

```
SELECT    *
FROM      STUDENT
WHERE     STMOBILE IS NULL ;
```

Example 11: SQL uses the logical operators, i.e., NOT, AND and OR. The precedence of these operators is shown in the following table:

Operators in increasing order of Precedence
Comparison operators (e.g. <, =, > etc.)
NOT
AND
OR

To print the name of all the students of PGDCA or BCA can also be written as:

```
SELECT    STID, STNAME
FROM      STUDENT
WHERE     PROGCODE =  "BCA" OR PROGCODE = "PGDCA" ;
```

Ordering of Results using ORDER BY clause

The SELECT statement of SQL, in addition to SELECT, FROM and WHERE clauses, supports three more clauses, viz. ORDER BY clause, GROUP BY clause and HAVING clause. We will discuss the ORDER BY clause here and the remaining two clauses are discussed in the next section.

The ORDER BY clause is used, when you want to display the records in the sorted sequence of one or more columns. This sorted order can be increasing or decreasing. Also, you can use more than one column for sorting the data. Please note the following points about ORDER BY clause:

- ORDER BY is the last clause of a SELECT statement.
- You can mention ASC for ascending order or DESC for descending order. In case you do not mention any order, then ordering, by default, is in ascending order.
- You can sort the records on a column, which is part of the table given in FROM statement, even if you have not displayed that column.
- You can also sort on the alias that has been created by you in the select statement.

Example 12: List the name of the students in the order of programme and within a programme in alphabetical order.

```
SELECT    PROGCODE, STNAME
FROM      STUDENT
ORDER BY  PROGCODE, STNAME ;
```

Check Your Progress 2

1) Write the SQL commands to insert the following data in the STUDENT and PROGRAMME table. Highlight the errors, if any.

Relation Name	PROGRAMME	
PROGCODE	PROGNAME	FEE
BCA	Bachelor of Computer Applications	48000
MCA	Master of Computer Applications	60000

Relation Name	STUDENT		
STID	STNAME	PROGCODE	STMOBILE
1002	Abc	BCA	
1003	NULL	PGDCA	91999999999
1004	XYZ	MCA	

……

……

……

2) List the various clauses of the SELECT statement giving their purpose.

……

……

3) Consider the following two relations – S and SP.

S

SupplierNo	SupplierName	SupplierRanking	SupplierCity
S1	ABC	60	Delhi
S2	DEF	30	Mumbai
S3	EFG	50	Bangaluru
S4	FGH	40	Chennai
S5	GHI	NULL	Delhi

SP

SupplierNo	PartNo	QuantitySold
S1	PA	1000
S2	PA	500
S2	PB	1200
S3	PB	700
S5	PA	2100
S5	PB	1200

a) Find the name of the suppliers, whose ranking is lower than 40 and the city of the supplier is Chennai.
b) List the names of the suppliers of Delhi city in the decreasing order of the supplier ranking.
c) List the names of the supplier who have supplied the part PB (use IN operator).
d) List the codes of all the suppliers who have supplied at least one part.

e) List all the suppliers, who have "EF" in their names.
f) Get part numbers for parts whose quantity sold in a supply is greater than 1000 or are supplied by S2. (Hint: It is a retrieval using union).
g) List the names of the suppliers, whose ranking is not given.

……
……..

7.5 GROUP BY CLAUSE AND AGGREGATE FUNCTIONS

In the previous section, you have gone through the concept of data manipulation language (DML). We have discussed the SELECT statement and its various clauses. In a database system, several queries require DBMS to produce information about a group. For example, you may be interested in finding the average marks of the group of PGDCA students vis-à-vis BCA students. The SQL supports a GROUP BY clause for such cases. In addition, SQL also supports a number of functions that can find aggregate information for a group of records. These functions are required to find the sum, average, counting of records etc. These are called aggregate functions. The following table defines some of the important aggregate functions used in SQL.

count	Used to count the number of records
sum	Finds the sum of the data of a column
avg	Finds the average of the data of a column
max	Finds the maximum value from the data of a column
min	Find the minimum value from the data of a column

Figure 3: Some Aggregate functions in SQL

Let us explain the use of these functions with the help of a few examples.

Example 13: Find the number of students of PGDCA.

```
SELECT      COUNT(*)
FROM        STUDENT
WHERE       PROGCODE = "PGDCA";
```

Please note that the wildcard *, in this case, indicates all the records to be counted, which fulfil the WHERE condition.

Example 14: Find the minimum, maximum and average fees of all the programmes.

```
SELECT      MIN(FEE), MAX(FEE), AVG(FEE)
FROM        PROGRAMME
```

GROUP BY clause

GROUP BY clause can be used to group records on certain criteria, e.g., you can group students on the basis of their Programmes. This clause is added after WHERE clause in the SELECT statement. In a SELECT statement in which you have used a GROUP BY clause, you can only use the aggregate functions or the column name on which you have grouped the data in the SELECT clause. The following example demonstrates this aspect:

Example 15: Find the number of students in every programme of the University.

You can answer this query by grouping the records on PROGCODE and finding the count of the student ids in each group.

```
SELECT      PROGCODE, COUNT(STID)
FROM        STUDENT
GROUP BY    PROGCODE;
```

Please note that in the SELECT clause of the SELECT statement above, we have used only the PROGCODE and the aggregate function COUNT.

HAVING clause

The HAVING clause can be used to specify a condition for a group. It is different from the WHERE clause, which is applicable for all the records, whereas the condition specified in the HAVING clause is to be fulfilled by a group of data. The following example explains the use of the HAVING clause.

Example 15: Count the number of students in each programme, where the mobile number of students is not given. List only those programmes which have more than 5 such Students.

```
SELECT      PROGCODE, COUNT(STID)
FROM        STUDENT
WHERE       STMOBILE IS NULL
GROUP BY    PROGCODE
HAVING      COUNT(STID) < 5;
```

Please note that in the SELECT clause of the SELECT statement above, we have used only the PROGCODE and the aggregate function COUNT.

7.6 DATA CONTROL LANGUAGE

The purpose of data control language (DCL) is to create users and assign access rights to them. In general, these commands are executed by a database administrator. The following are some of the most used DCL commands.

Creating a new user: You can create a new user using the following command:

```
CREATE USER < username for database user> IDENTIFIED BY < Password for the user>
```

For example, you can create a new user with the username "PGDCA_Student" with the password "PGDCA123"

```
CREATE USER PGDCA_Student IDENTIFIED BY PGDCA123
```

Use of GRANT Command: GRANT is used to give different kinds of accesses to a database user. Block3 covers the basic aspects of the GRANT option. In general, SQL supports two kinds of access permissions:

- The permissions that are at the system level.
- The permissions at the level of an object, such as a table, record, column etc.

The system-level permissions are, in general, specific to the DBMS environment, therefore, you may refer to the system documentation for details on such permissions. In this section, we provide a basic introduction to object-level permissions with the help of examples.

To give permission to get information from a table, the following SQL command may be used.

```
GRANT SELECT ON STUDENT, PROGRAMME TO PGDCA_Student;
```

To give permission for inserting and updating a record in the STUDENT table, you may use the following SQL command:

GRANT INSERT, UPDATE ON STUDENT TO PGDCA_Student;

In case you want to GRANT the SELECT access rights to more than one user on STUDENT table, then you may use the following command:

GRANT SELECT ON STUDENT TO PGDCA_Student1, PGDCA_Student2;

Use of REVOKE command: The REVOKE command is used to remove the access permissions that were granted to a user.

For example, you can also revoke SELECT permission using the following command:

REVOKE SELECT ON STUDENT FROM PGDCA_Student;

The following command will revoke all permissions of a user on STUDENT table.

REVOKE ALL ON STUDENT FROM PGDCA_Student;

Use of DROP command: You can remove a user using the DROP command**.**

For example, you can drop PGDCA_Student using the following SQL command:

DROP USER PGDCA_Student;

Check Your Progress 3

Consider the supplier relations given in Question 3 of Check Your Progress 2.

1) Write the SQL commands for the following aggregate operations.
   a) Find the total supply of part PA.
   b) Count the number of suppliers who have made a supply.
   c) List the part number and the total quantity supplied for that part.

……

……

2) Write the SQL commands for the following queries:
   a) List the suppliers who have made more than one supply.
   b) List the suppliers who have made more than one supply, with each supply being more than 1000.
   c) Find the part number and maximum quantity supplied, for the parts for which the total quantity supplied for that part is more than 1000. You may order the result in ascending order of part numbers.

……

……

3) Write the SQL commands for the following:
   a) Create a new user ABC having the password XYZ.
   b) Grant the user ABC to read table S only.
   c) Cancel all the access permissions of ABC on table S.

……
……

7.7 SUMMARY

SQL is one of the most important languages for any RDBMS. It allows a user to create tables, insert and update data in the tables and retrieve information from the tables. In addition, data of a table can be made available to only authorised users. This unit first introduces you to basic aspects of SQL and thereafter discusses the DDL commands. It is important that while creating tables you also include the constraints that are to be fulfilled by the tables. The constraints in SQL may be implemented using CHECK clause, PRIMARY KEY clause, FOREIGN KEY clause etc. You must also implement the referential action in case of referential integrity. You can also alter the tables if needed. This unit discusses the DDL commands for all the above operations. After creating the table using the DDL commands, next you use the DML commands to insert data into the tables. In case of any changes in the data values in the table, you may use the UPDATE command of DML. The commands for data insertion and updating have been discussed in the unit. One of the most important DML commands is SELECT which allows the retrieval of data from the table based on various criteria. This unit discusses various clauses of SELECT statement, viz., SELECT, FROM, WHERE, GROUP BY, HAVING and ORDER BY clauses. Finally, the unit discusses the CREATE USER, GRANT, REVOKE and DROP commands of DCL. You may please note that this unit does not cover all the SQL commands. You must practice these commands and learn more SQL commands from the user documentation of DBMS that you may use.

7.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) The following are the advantages of using SQL:
   - It is a standard query language across RDBMSs; therefore, you can port your data from one DBMS to another. Also, you just need to master SQL to use any RDBMS.
   - SQL is more English-like, so is easy to learn and use.
   - SQL exists for both interactive environments and as embedded SQL. Thus, you can use SQL for ad-hoc queries as well as in programs, like web-based programs.

   Some of the disadvantages of SQL are:
   - A query can be implemented in many ways, this may confuse a user.
   - The language, in its present form, is very large.
   - Some of the functions of the SQL are not portable across DBMS.
   - The result of an SQL command may allow duplicates, which are not relational in nature.

2) In this implementation, a lot of constraints have been defined, so instead of directly putting them in tables, first, we define the domains and use these domains in the CREATE table command. This will be a neater and more maintainable implementation.
   CREATE DOMAIN TYPEOFROOM AS CHAR (1) CHECK (VALUE IN (S, D));

```
// This makes the domain based on the first constraint. S represents a single room and D
represents a double room.
CREATE DOMAIN ROOMRENT AS DECIMAL(5, 0)
                    CHECK( VALUE BETWEEN 5000 AND 25000);
// Implements the second constraint
CREATE DOMAIN ROOMNUMBER AS INTEGER (3)
                    CHECK (VALUE  > = 101 AND  VALUE <=151);
//Implements the third constraint. Now you are ready to create the Tables.
CREATE TABLE ROOM (
        RNo         ROOMNUMBER      PRIMARY KEY,
        RType       TYPEOFROOM      NOT NULL DEFAULT S,
        RRent       ROOMRENT        NOT NULL,
        );
// Default room type stated in the first constraint is implemented in this statement.
// Next, we create the CUSTOMER table, BOOKING table will have references to both ROOM and
CUSTOMER tables.
CREATE TABLE CUSTOMER (
        CustID      CHAR(5)         PRIMARY KEY,
        CustName    VARCHAR(40)     NOT NULL,
        CustAddress VARCHAR(60)     NOT NULL
        CustPhone   CHAR (12)       NOT NULL
);
CREATE DOMAIN DATEOFBOOKING AS DATETIME
        CHECK (VALUE > = GETDATE() );
// Please note that GETDATE() function will output the current date.
// For a different RDBMS, this function may be different.
CREATE TABLE BOOKING (
        CustID      CHAR(5)         NOT NULL,
        RNo         ROOMNUMBER      NOT NULL,
        BookedFrom  DATEOFBOOKING   NOT NULL,
        BookedTo    DATEOFBOOKING   NOT NULL,
        PRIMARY KEY (CustID, RNo, BookedFrom),
        FOREIGN KEY (RNo) REFERENCES ROOM
                ON DELETE RESTRICT ON UPDATE CASCADE,
        FOREIGN KEY (CustID) REFERENCES CUSTOMER
                ON DELETE RESTRICT ON UPDATE CASCADE,
        CONSTRAINT BOOKINGDATEVERIFICATION
                CHECK ( NOT EXISTS ( SELECT *
                                FROM BOOKING X BOOKING Y
                                WHERE X.RNo = Y.RNo AND
                                X.BookedFrom > Y.BookedFrom AND
                                X.BookedFrom < Y.BookedTo)
                        );
```

//Please note this constraint may not work on certain DBMS, where NOT EXISTS clause is not supported.

Check Your Progress 2

1) You may use the following sequence of instructions:

INSERT INTO PROGRAMME
VALUES ("BCA", "Bachelor of Computer Applications", 48000);

The above insertion will be successful.

INSERT INTO PROGRAMME
VALUES ("MCA", "Master of Computer Applications", 60000);

The above insertion will fail, as the FEE is more than the allowable fee value.

INSERT INTO STUDENT (STID, STNAME, PROGCODE)
VALUES ("1002", "Abc", "PGDCA");

The above insertion will be successful, as PGDCA was inserted in Section 7.4.1

INSERT INTO STUDENT (STID, PROGCODE, STMOBILE)
VALUES ("1003", "BCA", "91999999999");

The above insertion will fail, as the Student name column value must be given, as it has a constraint NOT NULL.

INSERT INTO STUDENT (STID, STNAME, PROGCODE)
VALUES ("1003", "XYZ", "MCA");

The above insertion will fail, as the MCA programme insertion had failed earlier, so it will result in a violation of the referential constraint.

2) The following are the basic clauses of SELECT statements.

SELECT is used to specify the columns or expressions that are to be displayed in the result.

FROM is used to list the tables that are to be used for data output.

WHERE is used to specify conditions for the selection of data for the display.

GROUP BY clause is used to specify the columns, whose values should be used to group the records of the table.

HAVING is used to specify conditions, which should be fulfilled by each group for selection for display.

ORDER BY is used to specify the ordering of the records for the output.

3) a)
```
SELECT SupplierName
FROM S
WHERE SupplierRanking < 40 AND  SupplierCity = "Chennai";
```
The output of this will be:

SupplierName
FGH

b)
```
SELECT SupplierName
FROM S
WHERE SupplierCity = "Delhi"
```

ORDER BY SupplierRanking DESC;
The output of this will be:

SupplierName
ABC
GHI

c) SELECT SupplierName
FROM S
WHERE SupplierNo IN (SELECT DISTINCT SupplierNo
FROM SP
WHERE PartNo= “PB”) ;

The output of this will be:

SupplierName
DEF
EFG
GHI

d) SELECT DISTINCT SupplierID
FROM SP;
The output of this will be:

SupplierNo
S1
S2
S3
S5

e) SELECT *
FROM S
WHERE SupplierName LIKE %EF% ;
The output of this will be:

SupplierNo	SupplierName	SupplierRanking	SupplierCity
S2	DEF	30	Mumbai
S3	EFG	50	Bangaluru

f) SELECT DISTINCT PartNo
FROM SP X
WHERE X.QuantitySold > 1000
UNION
(SELECT DISTINCT PartNo
FROM SP Y
WHERE Y.SupplierNO = “S2”) ;
The output of this will be:

PartNo

PA
PB

g) SELECT SupplierName
FROM S
WHERE SupplierRanking IS NULL ;
The output of this will be:

SupplierName
GHI

Check Your Progress 3

1)

a) SELECT sum(QuantitySold)
FROM SP
WHERE PartNo = “PA” ;

b) SELECT count(DISTINCT SupplierNo)
FROM SP;

c) SELECT PartNo, sum(QuantitySold)
FROM SP
GROUP BY PartNo

2)

a) SELECT SupplierNo
FROM SP
GROUP BY SupplierNo
HAVING Count(PartNo) > 1 ; // As each supply is of one part only.

b) SELECT SupplierNo
FROM SP
WHERE QuantitySold > 1000
GROUP BY SupplierNo
HAVING Count(PartNo) > 1 ; // As each supply is of one part only.

c) SELECT PartNo, Max(QuantitySold)
FROM SP
GROUP BY PartNo
HAVING Sum(QuantitySold) > 1000
ORDER BY PartNo;

3)

a) CREATE USER ABC IDENTIFIED BY XYZ;
b) GRANT SELECT ON S TO ABC;
c) REVOKE ALL ON S FROM ABC;

UNIT 8 Structured Query Language (Part-II)

Structure

8.0 INTRODUCTION

In the previous unit, you have gone through the details of the basic set of commands of the Structured Query Language (SQL). A good database system consists of normalised relations, which involve the lossless decomposition of relations into smaller relations. These smaller relations have controlled redundancy and help in removing the redundancy-related problems of a database, which you have studied in the earlier units of this course. However, when you want to retrieve information from a database, you may be required to extract information from several smaller relations. Join is one of the most important operations, which can be used to provide information from multiple tables. This unit discusses different types of join operations that can be performed on database tables.

In many database queries, you may be required to use a sub-query that provides data for the main query. This unit discusses the subqueries and correlated subqueries. The unit also introduces you to the concept of views, sequences, indexes and synonyms database objects. You must practice these queries, to master them.

8.1 OBJECTIVES

After going through this unit, you should be able to:

- Define different kinds of joins that can be performed on tables;
- Write the SQL statements for Join commands;
- Use subqueries and correlated subqueries;
- Write nested subqueries;
- Define and write SQL commands to create views and indexes;
- Define and write SQL commands for creating sequences and synonyms.

8.2 SQL JOINS

The join is an important relational operation for queries that involve more than one relation. You have already gone through the relational join operation in an earlier unit, this unit explains the use of the join operation in SQL. The purpose of join operation is to combine the data from two different tables on specific attribute(s), which share the same domain. *The necessary condition for the meaningful join of two tables is that each of these tables should have at least one attribute, called the joining attribute, that should have the same data domain*. The join operation uses the joining condition to combine the data of the two tables. In this unit, we will use the tables and data given in Figure 1 for various examples of join operations.

Table Name: CLIENT			
ClientID	ClientName	ClientAddress	ClientPhone
C001	ABCD	79, MGRoad	9999999991
C002	BCDE	Raman Street	9999999992
C003	CDEF	Vindyachal apatments	9999999993

Table Name: ITEM			
ItemID	ItemName	ItemPrice	QuantityAvailable
I01	Pen	20	500
I02	Pencil	5	1000
I03	Paper Sheet	20	1000
I04	Sharpener	5	200

Table Name: ORDER	
OrderID	ClientID
O001	C001
O002	C002
O003	C001
O004	C003
O005	C002

Table Name: ORDERDETAILS		
OrderID	ItemID	QuantityPurchased
O001	I02	2
O001	I03	10
O001	I04	1
O002	I02	5
O003	I01	2
O003	I03	5
O004	I01	5
O004	I03	3
O005	I01	8

Figure 1: Sample Tables and Data

The tables above do not have a date of purchase, as these tables are just for demonstrating various commands. Please note that the primary key of each table has been underlined. Please also note that each OrderID is unique, and an order may contain several items. The foreign keys in the tables are:

- OrderID in ORDERDETAILS table references ORDER table
- ItemID in ORDERDETAILS table references ITEM table
- ClientID in ORDER table references CLIENT table

Now, let us answer the following questions about the join operation:

1. Can you join the CLIENT and the ITEM tables? No, as there is no common attribute in these two tables on which table can be joined. Please note that two tables can be joined only if they have at least one attribute each which has the same domain.
2. How will you find the name of items that have been ordered by a client, whose name is ABCD? ClientID is C001? In order to answer it first let us assume that your DBMS supports the clause

```
SELECT * INTO # <Name of a Temporary Table>
FROM …;
```

Now, you can find the relevant information in the following way (a very cumbersome way):

a) First, find the ClientID of the client whose name is "ABCD" from the CLIENT table using the command:

```
SELECT ClientNo INTO  #TempCLIENT
FROM CLIENT
WHERE ClientName = "ABCD" ;
```

The output of this query would be a Temporary table named #TempCLIENT. As per the data of Figure 1, the only ClientID with the name ABCD is C001. Thus, the output of the command above would be:

Table Name: #TempCLIENT
ClientID
C001

b) Next, you can find the orders made by ClientID C001 from the ORDER table using the command:

```
SELECT OrderID INTO  #TempORDER
FROM ORDER
WHERE ClientID IN (     SELECT *
                        FROM #TempCLIENT);
```

The output of the command above would be:

Table Name: #TempORDER
OrderID
O001
O003

c) Next, you will find the items that have been ordered in each of the OrderID O001 and O003 from the ORDERDETAILS table using the command:

```
SELECT DISTINCT ItemID INTO  #TempORDERDETAILS
FROM ORDERDETAILS
WHERE OrderID IN (      SELECT *
                        FROM #TempORDER);
```

The output of the command above would be:

Table Name: #TempORDERDETAILS
ItemID
I02
I03
I04
I01

d) Now, you can find the name of the ordered items by C001, from the ITEM table using the following command:

```
SELECT ItemID, ItemName
FROM ITEM
WHERE ItemID IN (       SELECT *
                        FROM #TempORDERDETAILS);
```

The output of the command above would be:

ItemID	ItemName
I02	Pencil
I03	Paper Sheet
I04	Sharpener
I01	Pen

Figure 2: The Output of the Command

Can this query be answered with a single query? You may require the JOIN command to solve this query.

```
SELECT DISTINCT  i.ItemID, i.ItemName
FROM     CLIENT c, ORDER o, ORDERDETAILS oo, ITEM i
WHERE    c.ClientName = "ABCD" AND
         c.ClientID = o.ClientID AND
         o.OrderID = oo.OrderID  AND
         oo.ItemID = i.ItemID
```

Though the statement above looks very complicated, you can interpret it by the following statements:

- In the select statement, you have put the information that you want to output. The ItemName information is available in only ITEM table, whereas ItemID is available both in ITEMDETAILS and ITEM tables, thus, the alias i before the ItemID is necessary and informing the database server that ItemID data is to be obtained from the ITEM table (please note i is an alias to ITEM table.).
- In the FROM clause have used aliases for every table that is being used.
- The WHERE clause has the following conditions:
    - c.ClientName = "ABCD", which will identify the ClientID as C001.
    - c.ClientID = o.ClientID will join the CLIENT table (alias c) with Order table (alias o), and output O001 and O003.
    - o.OrderID = oo.OrderID will join the ORDER and ORDERDETAILS tables
    - oo.ItemID = i.ItemID will join the ORDERDETAILS and ITEM tables.

  Thus, producing the final result, as shown in Figure 2.

This example may seem to be very complex, but you must once again go through the example after going through the remaining part of this unit. Now, let us now focus on different kinds of joins.

8.2.1 Equi-join

Equi-join, as the name suggests, has equality as the joining condition. This means that the attributes that you are using to join would be checked for equality. Please note that the names of the joining columns in the two tables may be different. The following is an example of an equijoin operation:

Example 1: Using the tables of Figure 1answer the query:

List the ID, name of the client, and the order ids of all the clients who have placed an order.

You need to use the CLIENT and ORDER tables to answer this query, as the name of the clients are in the CLIENT table and the order id information is in the ORDER table. You may use equi-join operation for this query using the columns ClientNo in CLIENT and ClientNo in the ORDER table. Though in this present example, both the columns have the same name, i.e. ClentID, equi-join operation does not require these names to be the same. SQL query for the query is presented below:

Query Using equi-join:

```
SELECT c.ClientID, c.ClientName, o.ClientID, o.OrderID
FROM CLIENT c, ORDER o
WHERE c.ClientID = o.ClientID;
```

The result of the query, on the tables of Figure 1, will be:

c.ClientID	ClientName	o.ClientID	OrderID
C001	ABCD	C001	O001
C001	ABCD	C001	O003
C002	BCDE	C002	O002
C002	BCDE	C002	O005
C003	CDEF	C003	O004

The output of shows the ClientID of both the CLIENT and the ORDER tables. We can display any one of these two columns. Please also note that joining may result in the display of redundant information, as you can observe in the Client name column.

Natural Join: Natural join is one of the most used join operations. It requires that the joining columns of the two tables should have the same name. It uses the equality condition. The result of the join operation displays only one of the joining attributes/columns.

The query of example 1 can also be answered using the natural join operation, as both the tables have the same column name ClientID. The following SQL command will perform the natural join operation:

```
SELECT  CLIENT.ClientID, ClientName, OrderID
FROM CLIENT NATURAL JOIN ORDER;
```

The output of the command would be:

CLIENT.ClientID	ClientName	OrderID
C001	ABCD	O001
C001	ABCD	O003
C002	BCDE	O002
C002	BCDE	O005
C003	CDEF	O004

Inner Join: Inner join combines two tables using a combining condition on the joining attributes. In the present versions of SQL, you may have to use INNER JOIN command for example 1 query:

```
SELECT CLIENT.ClientID, ClientName, OrderID
FROM CLIENT INNER JOIN ORDER
        ON CLIENT.ClientID = ORDER.ClientID;
```

8.2.2 Outer Join

Consider the situation that a new client has been added to the client table and the present state of the CLIENT table is:

Table Name: CLIENT			
ClientID	ClientName	ClientAddress	ClientPhone
C001	ABCD	79, MGRoad	9999999991
C002	BCDE	Raman Street	9999999992
C003	CDEF	Vindyachal apatments	9999999993
C004	DEFG	Maidan Road	9999999994

Further, assume that this new client has not issued any order. Let us now issue the query of example 1 again on the present state of the database. You will find the result of the query will still be the same, as shown in example 1. So, we get no information about C004, in the result of the query.
If the objective of the query was to show the list of all the clients and their OrderIDs, irrespective of the fact, that they have given zero or more orders, then how would you extract such information? In effect, you want that all the records from the CLIENT table should participate in joining even if there is no joining row in the ORDER table. This requires the use of OUTER JOIN operation.

Example 2: List all the clients and their orders (NULL in case no order is given by a client).
The following SQL command will be required for the query asked above:

```
SELECT CLIENT.ClientID, ClientName, OrderID
FROM CLIENT LEFT OUTER JOIN ORDER
        ON CLIENT.ClientID = ORDER.ClientID;
```

We have used the term LEFT OUTER JOIN, as in this command we want that all the records of the table mentioned on the left side of the join command (CLIENT in this case) be part of the output, irrespective of a joining record on the table on the right side (ORDER in this case). The output of the command would be:

CLIENT.ClientID	ClientName	OrderID
C001	ABCD	O001
C001	ABCD	O003
C002	BCDE	O002
C002	BCDE	O005
C003	CDEF	O004
C004	DEFG	NULL

You may notice that the last record in this table is for Client C004, who has not placed any order yet. This record is displayed as we have used the Left Outer join operation. Similarly, you can use the RIGHT OUTER JOIN or FULL OUTER JOIN, as per the need of database output.

8.2.3 Self-Join

In the self-join operation, a table is joined with a copy of itself. This is very useful for queries, which draw information from within a column of a table. The following is an example of the self-join.

Example 3: Find the pairs of similar priced items in the ITEM table.
You may first simply inspect the ITEM table. You will find that pairs -Pen and Paper Sheet; and Pencil and Sharpener, have the same price. How did you find this information? You compared the ItemPrice of each item with other ItemPrice. This inspection, in general, can be performed by join operation. Hence, you can answer the query using the following SQL command:

```
SELECT i1.ItemName, i2.ItemName
```

FROM ITEM i1, ITEM i2
WHERE i1.ItemPrice = i2.ItemPrice;

Thus, you are joining the two tables on identical ItemPrice, and displaying the pair of names of the items that have similar prices. However, you will get the following output of this SQL command:

i1.ItemName	i2.ItemName
Pen	Pen
Pen	Paper Sheet
Pencil	Pencil
Pencil	Sharpener
Paper Sheet	Paper Sheet
Paper Sheet	Pen
Sharpener	Sharpener
Sharpener	Pencil

You may observe that the output of the command has many extra records like records with the same item, e.g. the pair (Pen, Pen). In addition, a pair like (Pen, Paper Sheet) is same as (Paper Sheet, Pen), therefore, only one of these pairs should appear in the result. A simple solution to this problem is to add another condition in the SQL command: (i1.ItemID < i2.ItemID). This will ensure that only those pairs in which ItemID of first table is less than ItemID of second table will be part of output, thus, eliminating both the problems, as stated above. The command is as follows:

SELECT i1.ItemName, i2.ItemName
FROM ITEM i1, ITEM i2
WHERE i1.ItemPrice = i2.ItemPrice AND i1.ItemID < i2.ItemID;

Thus, you are joining the two tables on identical ItemPrice, and displaying the pair of names of the items that have similar prices. You will get the following output of this SQL command:

i1.ItemName	i2.ItemName
Pen	Paper Sheet
Pencil	Sharpener

Check Your Progress 1

Consider the Tables, given in Figure 1, and answer the following questions:

1) (a) Find the list of those item names that have been supplied, as part of at least one Order.
(b) List the order details including the order id, item code, item name and price.
(c) Compute the total price of an order.

……

……

……

……..

2) Find the list of all the item IDs, item names and related Order IDs. This query should list the name of the items even if it is not part of any order.

……

……

……

3) Find the list of items that have been purchased together.

……

……

……

8.3 NESTED QUERIES

In the previous section, we discussed join queries, which are responsible for joining the data from two tables. Nested queries are used when the output of a query may be useful for the execution of another query. Thus, you can create a main query nest a sub-query in any of the clauses of this main query. In this section, we discuss the basic type of nested queries and then a special type of nested subqueries called correlated subqueries.

8.3.1 Sub-queries

A sub-query is another SELECT statement that is used in the main SELECT statement. Please remember the following points about a subquery:

- A sub-query is executed prior to the main query. Therefore, you can use the result of a sub-query in an expression of the main query.
- A sub-query, on its execution, can return either a single value or a set of values or a relation. Therefore, the result of the sub-query can be used in a comparison in the WHERE or HAVING clause or for comparison with a set operator; or even in the FROM clause when a relation is returned.
- You can put a sub-query inside a sub-query. You can use a separate table in the main and sub-query.
- You should not use the ORDER BY clause in a sub-query, rather it should be used as the last clause of the main query. However, you can use the GROUP BY clause in the sub-query.
- When you use sub-queries in the WHERE or HAVING clause of the main query, you may be required to use comparison or set operators. Some of these operators are given below:

Comparison Operators	<, =, >=, <, <=, <>	Used when the sub-query returns a single constant value
Set Operators	IN, ANY, ALL	Used when sub-query returns a set of values.

These operators will be explained in various examples. We will use tables and data from Figure 1 to answer the following queries.

Example 4: Find the item, whose available quantity is the smallest.

```
SELECT        *
FROM          ITEM
WHERE         QuantityAvailable  =  ( SELECT       MIN (QuantityAvailable)
                                      FROM         ITEM);
```

The sub-query in this case will find a single minimum value, which will be compared in WHERE clause to determine the record, which has that minimum value.

ItemID	ItemName	ItemPrice	QuantityAvailable
I04	Sharpener	5	200

Example 5: Find the item, whose price is the minimum.

```
SELECT      *
FROM        ITEM
WHERE       ItemPrice = (       SELECT      MIN (ItemPrice)
                                FROM        ITEM);
```

The sub-query in this case has found a single minimum value, which will be compared in the WHERE clause. Please note that in this case, the output contains two records.

ItemID	ItemName	ItemPrice	QuantityAvailable
I02	Pencil	5	1000
I04	Sharpener	5	200

Example 6: Find the items, whose price is greater than the price of a Pencil and the quantity available is more than the Item whose ItemID is I01. Assume that item names are unique.

```
SELECT      *
FROM        ITEM
WHERE       ItemPrice >     (       SELECT      ItemPrice
                                    FROM        ITEM
                                    WHERE       ItemName = "Pencil")
            AND
            QuantityAvailable > (   SELECT      QuantityAvailable
                                    FROM        ITEM
                                    WHERE       ItemID = "I01");
```

There are two sub-queries in this case, the output of the first sub-query would be value 5 and the output of the second sub-query would be value 500. The result of the query is given below:

ItemID	ItemName	ItemPrice	QuantityAvailable
I03	Paper Sheet	20	1000

Example 7 (a): Find the total quantity of all the items in every order.

This query can be answered by the ORDERDETAILS table using a simple GROUP BY clause:

```
SELECT      OrderID, SUM(QuantityPurchased) AS TotalofAllItems
FROM        ORDERDETAILS
GROUP BY    OrderID;
```

OrderID	TotalofAllItems
O001	13
O002	5
O003	7
O004	8
O005	8

Example 7 (b): Find the total quantity of all the items for the orders, which have ordered more total quantity than ordered in order O003.

```
SELECT      OrderID, SUM(QuantityPurchased) AS TotalofAllItems
FROM        ORDERDETAILS
GROUP BY    OrderID
HAVING      SUM(QuantityPurchased) > (  SELECT SUM(QuantityPurchased)
                                        FROM ORDERDETAILS
                                        WHERE OrderId= "O003");
```

Please note that the sub-query in this case is in the HAVING clause. This command will output:

OrderID	TotalofAllItems
O001	13
O004	8
O005	8

Example 8(a): Find the total price of each Order.

This query would require joining of ORDERDETAILS table and ITEM table, as the details of an order are in the ORDERDETAILS and Price of each order is in ITEM table. The following command would simply compute the Price of each Item of the order:

```
SELECT ORDERDETAILS.*, ItemName, ItemPrice, (QuantityPurchased*Price) AS ItemTotal
FROM  ORDERDETAILS, ITEM
WHERE ORDERDETAILS.ItemID = ITEM.ItemID;
```

This command will list the orders as under:

OrderID	ORDERDETAILS.ItemID	QuantityPurchased	ItemName	ItemPrice	ItemTotal
O001	I02	2	Pencil	5	10
O001	I03	10	Paper Sheet	20	200
O001	I04	1	Sharpener	5	5
O002	I02	5	Pencil	5	25
O003	I01	2	Pen	20	40
O003	I03	5	Paper Sheet	20	100
O004	I01	5	Pen	20	100
O004	I03	3	Paper Sheet	20	60
O005	I01	8	Pen	20	160

To find the answer of the stated query, you need to find the sum of item total for each order. Thus, you can answer of the stated query by the following SQL command:

```
SELECT OrderID, SUM(QuantityPurchased*Price) AS OrderTotal
FROM  ORDERDETAILS, ITEM
WHERE ORDERDETAILS.ItemID = ITEM.ItemID
GROUP BY OrderID;
```

This command will list the orders as under:

OrderID	OrderTotal
O001	215
O002	25
O003	140
O004	160
O005	160

Example 8(b): Find the total price of the orders, which have at least one order for the Item whose item name is Paper Sheet.

This query would require you to use the answer of the query 8(a) and use of join in the subquery. The answer to the subquery would be a set of records, therefore, we will use set operator IN. The query is given as follows:

```
SELECT OrderID, SUM(QuantityPurchased*Price) AS OrderTotal
FROM  ORDERDETAILS, ITEM
WHERE ORDERDETAILS.ItemID = ITEM.ItemID AND OrderID IN
                                                    (   SELECT DISTINCT OrderID
                                                        FROM ORDERDETAILS oo, ITEM i
                                                        WHERE oo.ItemID = i.ItemID AND
                                                                ItemName = "Paper Sheet")
GROUP BY OrderID;
```

This command will list the orders as under:

OrderID	OrderTotal
O001	215
O003	140
O004	160

Example 9: Find the id and name of all the clients, who have ordered Pen and Paper Sheet in a single order. This query would require the joining of the ORDER table and CLIENT table in the main query and the ORDERDETAILS table and ITEM table in the sub-query. The following command would find the list of OrderIDs, which have both Pen and Paper Sheet in a single Order.

```
(SELECT DISTINCT OrderID
FROM  ORDERDETAILS, ITEM
WHERE ORDERDETAILS.ItemID = ITEM.ItemID AND ItemName= "Pen")
        INTERSECT
( SELECT DISTINCT OrderID
FROM  ORDERDETAILS, ITEM
WHERE ORDERDETAILS.ItemID = ITEM.ItemID AND ItemName= "Paper Sheet");
```

This query will result in the following output:

OrderID
O003
O004

Now, you can write the query as follows:

```
SELECT DISTINCT (CLIENT.ClientID, ClientName)
FROM ORDER, CLIENT
WHERE ORDER.ClientID = CLIENT.ClientID AND
    OrderID IN (
        (SELECT DISTINCT OrderID
        FROM  ORDERDETAILS, ITEM
        WHERE ORDERDETAILS.ItemID = ITEM.ItemID AND ItemName= "Pen")
        INTERSECT
        ( SELECT DISTINCT OrderID
        FROM  ORDERDETAILS, ITEM
        WHERE ORDERDETAILS.ItemID = ITEM.ItemID AND ItemName= "Paper Sheet")
        );
```

This command will list the client's information as under:

ClientID	ClientName
C001	ABCD
C003	CDEF

Example 10: List the total price of the orders, whose total price is more than 150.

An alternate way of writing this query would be to use a subquery in the FROM clause, as given below:

```
SELECT OrderID, OrderTotal
FROM  (SELECT       OrderID, SUM(QuantityPurchased*Price) AS OrderTotal
       FROM         ORDERDETAILS, ITEM
       WHERE        ORDERDETAILS.ItemID = ITEM.ItemID
        GROUP BY OrderID)
WHERE OrderTotal > 150;
```

OrderID	OrderTotal
O001	215
O004	160

You may please note that an alias in the subquery is used in various clauses of the main query.

8.3.2 Correlated Subqueries

In many queries, the columns of the tables used in FROM clause are also used in the subquery. Such subqueries are called Correlated subqueries and generally are time-consuming queries. These queries are explained with the help of the following example.

Example 11: List the client id and name of the clients, who have ordered ItemID I01 and ItemID I03, as part of a single order.

This query can be answered as a correlated query as follows:

```
SELECT DISTINCT OrderID
FROM  ORDERDETAILS outer
WHERE ORDERDETAILS.ItemID = "I01" AND EXISTS
                ( SELECT DISTINCT OrderID
                  FROM ORDERDETAILS inner
                  WHERE outer.OrderID=inner.OrderID AND
                        outer.ItemID<inner.ItemID AND inner.ItemID = "I03"
                );
```

The execution of such a correlated query is time-consuming. Why? Since in these queries, the subquery is executed for each instance of the main query. For example, in the case of example 11, the main query will find that the order O003 fulfils the main clause. This will trigger the execution of the subquery, which will check that for the same value of order O003, there exists a record for item I03. Since it exists, therefore O003 will be output. Similarly, O004 will be output. In the case of O005, the main query has the record for I01, however, the record for I03 does not exist. Thus, O005 will not be output. Thus, this query will result in the following output:

OrderID
O003
O004

Check Your Progress 2

1) What is a subquery? When would you like to use the sub-query?

……

……

……

2) Consider the following relations:

EMPLOYEE (EmployeeNo, EmployeeName, EmployeePhone, EmployeeBasicPay)
PROJECT(ProjectNo, ProjectName, ProjectDuration)
EMPLOYEEPROJECT (EmployeeNo, ProjectNo, EmpRoleInProject)

Now answer the following queries:

(a) Find the name of the projects, which have the least duration.

(b) Find the name and basic pay of employees who are working on more than one project.

(c) Find the name of the employee who earns more than the average salary of all the employees.

……

……

……

3) Consider the following relations Schema given in question 2 above, and the following two SQL queries on this schema. What is the purpose of these two queries?

(i)
```
SELECT EmployeeName
FROM   EMPLOYEE
WHERE EmployeeNo IN ( SELECT EmployeeNo
                      FROM EMPLOYEEPROJECT
                      GROUP BY EmployeeNo
                      HAVING COUNT(EmployeeNo) =
                              (SELECT COUNT (*) FROM PROJECT));
```

(ii)
```
SELECT ProjectName
FROM PROJECT
WHERE ProjectNo  IN (( SELECT ProjectNo
                       FROM PROJECT)
                     MINUS
                     (SELECT DISTINCT ProjectNo
                      FROM EMPLOYEEPROJECT)
                     );
```

……..……

……..……

……..……

8.4 DATABASE OBJECTS

Database objects are useful concepts in a database system. In this section, we discuss four different types of objects, which are defined in many database management systems. Views are virtual tables, which may be used for implementing database security. In addition, they can also be used for database query optimisation. Sequences are used to maintain an automatic sequence of numbers, which can be very useful for input of unique values in a column. Indexes are used to enhance the performance of a database system. The following sub-section discusses these concepts in detail.

8.4.1 Views

A view is a part of external schema of a database, which can be used to restrict the data display, input and modification of a database system. A database system consists of base tables, which are the tables created for an entity or relationship of a logical or conceptual database model. The base tables are used to store data. A view can be categorised as a virtual table, which can be stored as a query on a table. This query is called the view definition. Please note that views do not store data of their own, rather data is stored in the base tables of the database. The query definition of a view can be stored in the data dictionary of DBMS. When you use a view, then it can be executed through the following steps:

- Retrieve the view definition from the data dictionary.
- Find the access privileges of the view, as well as the base tables associated with the view.
- Create the view qualified query, if the user has access privileges on the view and data of base tables.

The following are the Advantages of using views in a database system:

- Views allow database users to access only that data to which they have access privileges.
- Views mechanisms enforce logical data independence in a database system.
- Views allow different users to view the data differently.

How to create a view?

A view is created using a CREATE VIEW statement followed by a query. A view statement can include multiple tables in the FROM clause, the condition of joining two tables in the WHERE clause, the GROUP BY clause and expressions of a subquery. However, a view statement is not allowed to include an ORDER BY clause. The following are examples of view creation.

Example 12: Create a view for the logistics person, who would be interested in the list of all the items (with the item name and quantity) of a particular order, say O001, and the address of the client to whom that order is to be sent.

```
CREATE VIEW LOGISTICS AS
```

```
SELECT oo.OrderID OOID, oo.ItemID IItemID, ItemName, QunatityPurchased,
        c.ClientID CClientID, ClientName, ClientAddress, ClientPhone
FROM ORDERDETAILS oo, ITEM i, ORDER o, Client c
WHERE oo.ItemID = i.ItemID AND oo.OrderID = o.OrderID
                        AND o.ClientID=c.ClientID AND oo.OrderID= "O001";
```

This will create a view named LOGISTICS. you can display the content of the LOGISTICS view using the following command:

```
SELECT *
FROM LOGISTICS;
```

It will display the following output:

OOID	IItemID	ItemName	QuantityPurchased	CClientID	ClientName	ClientAddress	ClientPhone
O001	I02	Pencil	2	C001	ABCD	79, MGRoad	9999999991
O001	I03	Paper Sheet	10	C001	ABCD	79, MGRoad	9999999991
O001	I04	Sharpener	1	C001	ABCD	79, MGRoad	9999999991

Data Updates through Views?

Views can be used for data updates though there are several restrictions on updating data through the views. In general, you cannot update the data through a view, if it contains a JOIN operation or a GROUP BY clause or an aggregate function or a subquery. In addition, you may use a check option for views, which is defined next.

Creating views with check option: Consider a view that allows updates on data, but those updates may result in a modified record, which does not fulfil the view defining criteria. The following example explains this situation.

Example 13: Create a view of the items whose price is more than INR 15.

```
CREATE VIEW COSTLYITEM AS
SELECT *
FROM ITEM
WHERE ItemPrice > 15;
```

This will create the view consisting of two items I01 and I03. Assume that the price of the Pen is discounted and made Rs 10, if you allow an update through the COSTLYITEM view, then the price of item I01 (Pen) will be updated to INR 10. Thus, I01 will no longer be part of the view and disappear from the view.

The situation may be avoided if you use WITH CHECK OPTION, which will restrict such updates. Thus, you may use the following option to create the view:

```
CREATE VIEW COSTLYITEM AS
SELECT *
FROM ITEM
WHERE ItemPrice > 15
WITH CHECK OPTION;
```

8.4.2 Sequences

Consider a situation, where in a column, you want to assign the next number in a sequence in every new record. Sequences are used for generating a new unique number of a sequence for a specific column. Sequences may be used for automatically generating the unique primary key values, as they can be generated efficiently. The following example explains the use of sequences:

Example 14: You can create a sequence, say ENRSEQ, for generating unique enrolment numbers of the students as follows (Please note different DBMS may have a different command for such sequences):

```
CREATE SEQUENCE ENRSEQ
START WITH 230000001
INCREMENT BY 1
MAX VALUE 239999999;
```

The SQL commands, as given in example 14 will generate the sequence numbers ENRSEQ, but how will you use these sequences in tables? This is explained in example 15.

Example 15: This example demonstrates the use of sequence numbers while inserting records in a table. Consider a relation STUDENT (StudentID, StudentName, StudentProgCode). The StudentID is the primary key of the relation and should be unique and 9 characters long in the range 230000001 to 239999999. To automatically generate the sequence of student ids you may use the following command:

```
INSERT INTO STUDENT VALUES (ENRSEQ.NEXTVAL, "ABCD", "PGDCA");
INSERT INTO STUDENT VALUES (ENRSEQ.NEXTVAL, "BCDE", "BCA");
INSERT INTO STUDENT VALUES (ENRSEQ.NEXTVAL, "CDEF", "PGDCA");
```

You can display the content of the table after these insertions using the following commands:

```
SELECT * FROM STUDENT;
```

StudentID	StudentName	StudentProgCode
230000001	ABCD	PGDCA
230000002	BCDE	BCA
230000003	CDEF	PGDCA

You may please note that all the details of a sequence, like starting number, increment value and maximum values are stored and updated from time to time in the data dictionary. Further, certain situations like the rollback of an insert operation may result in gaps in the inserted values of a sequence.

Can you modify a sequence, once created? Yes, for this purpose, you may use the ALTER SEQUENCE statement of SQL. For example, you can change the increment value to 5 and the maximum value to 235000000 for the sequence ENRSEQ using the following command: the

```
ALTER SEQUENCE ENRSEQ
START WITH 230000001
```

INCREMENT 5
MAX VALUE 235000000;

You can also delete a sequence by giving the following command:

DROP SEQUENCE ENRSEQ;

8.4.3 Indexes and Synonyms

An index is part of the physical/internal schema of relational design. It is one of the important access structures of a database. Indexes help in enhancing the speed of the query processing. Every database has a basic set of indexes, like the primary key index, which are created automatically by the database management system. Additional indexes can be created by the database administrator, if needed.

You can create indexes using the following command:

Example 16: Consider the STUDENT relation, as given in Example 15. You need to create indexes for (i) To enhance the queries related to various programmes (ii) to list the students of a programme in the order of their names, for this case, you may like to create an index on programme and student name. You can use the following commands to create these indexes:

CREATE INDEX PROGCODE_IDX ON STUDENT (StudentProgCode);
CREATE INDEX PROG_STUDENT_IDX ON STUDENT (StudentProgCode, StudentName);

It may be noted that the information of the indexes of the tables is stored in the data dictionary of a DBMS.

Further, you may note that though an index may help in improving query response time, they increase the time of data insertion, modification, and deletion, as these operations require updating the contents of related indexes. In general, you cannot modify the structure of an index, however, you can delete an index. To remove an index, you may give the command:

DROP INDEX PROGCODE_IDX;

Several DBMSs allow creation of alternative links or names to the database items or objects. These alternative names are generally short names and used for convenient references. For example, a typical DBMS may allow the creation of short names using the following command:

Example 17: You may create the following synonym for the STUDENT tables of example 15.

CREATE SYNONYM st
FOR STUDENT;

You can remove a synonym by giving the following command:

DROP SYNONYM st;

Check Your Progress 3

1) Consider the following relations:

EMPLOYEE (EmployeeNo, EmployeeName, EmployeePhone, EmployeeBasicPay)
PROJECT(ProjectNo, ProjectName, ProjectDuration)
EMPLOYEEPROJECT (EmployeeNo, ProjectNo, EmpRoleInProject)

Create a detailed view of Projects consisting of Project number, project name, employee name of employees working on the project and his/her role in the project.

……

……

……

2) Create a three-digit long odd number sequence.

……

……

3) What are the advantages of creating indexes?

……

……

8.5 SUMMARY

This unit introduces you to join queries and sub-queries. First, the unit explains the use of the JOIN statement in SQL. A JOIN statement uses two tables each of which has at least one attribute on the same domain. These attributes also called joining attributes, are used to join the tables using a conditional operator. In case, this conditional operator is equality, then the join operation is called the equi-join. In addition, if the joining attributes have the same name in both the tables, then only one of these attributes is included in the output. This type of join is called the Natural join. Outer joins are used when the records of the table, which do not participate in join operation are to be output too. Self-join is performed when one column values are to be related to the same or another column of the same table. The joining condition of self-join requires removing redundant records. The unit discusses these joins with the help of examples. Next, the unit discusses the nested queries. The unit details some of the simple operators that are used with sub-queries with the help of examples. This unit also discusses the correlated subqueries, which are generally expensive to execute. Finally, the unit provides details of some of the important concepts used in database management systems such as views, sequences and indexes.

You should practice these queries using a DBMS.

8.6 SOLUTIONS/ANSWERS

Check Your Progress 1

1) (a) The information about what parts have been supplied as a part of at least one order is available in the ORDERDETAILS table, it may be joined with the ITEM table to find the names of the parts.

```
SELECT DISTINCT ITEM.ItemID, ItemName
FROM ITEM, ORDERDETAILS
WHERE ITEM.ItemID = ORDERDETAILS.ItemID;
```

You can also use a subquery (explained in the next section) for this information:

```
SELECT ItemID, ItemName
FROM ITEM
WHERE ItemID IN (     SELECT DISTINCT ItemID
                      FROM ORDERDETAIL
);
```

(b) The information about what orders are available is in the ORDERDETAILS table and information about the items is available in the ITEM table. Therefore, you need to join these two tables for the required information.

```
SELECT OrderID, ORDERDETAILS.ItemID, ItemName, ItemPrice
FROM  ORDERDETAILS,  ITEM
WHERE ORDERDETAILS.ItemID = ITEM. ItemID;
```

(c) The following query may result in the desired information:

```
SELECT OrderID, SUM(QuantityPurchased*ItemPrice)
FROM  ORDERDETAILS,  ITEM
WHERE ORDERDETAILS.ItemID = ITEM. ItemID
GROUP BY OrderID;
```

2) It will require outer join:

```
SELECT ITEM.ItemID, ItemName, OrderID
FROM ITEM LEFT OUTER JOIN ORDERDETAILS
ON ITEM.ItemID = ORDERDETAILS.ItemID;
```

3) This will require a self-join.

```
SELECT od1.ItemID, od2.ItemID
FROM ORDERDETAILS od1, ORDERDETAILS od2
WHERE  od1.OrderID = od2.OrderID AND od1.ItemID < od2.ItemID;
```

Check Your Progress 2

1) A subquery is a SELECT statement that is part of a main SELECT statement. You may put the SELECT statement in the FROM, WHERE or HAVING clauses of the main SELECT statement. Subqueries are useful when you want to use the set operators. They can also be used where a smaller relation can be returned and used more efficiently.

2) (a) Find the name of the projects, which have the least duration. (Can you write a more efficient query in this case?)

```
SELECT ProjectName
FROM PROJECT
WHERE ProjectNo IN
        (SELECT MIN(ProjectDuration)
         FROM PROJECT );
```

(b) Find the name and basic pay of employees who are working on more than one project.

```
SELECT EmployeeNo, EmployeeName, EmployeeBasicPay
FROM EMPLOYEE
```

```
WHERE EmployeeNo IN
        (SELECT EmployeeNo)
         FROM EMPLOYEEPROJECT
        GROUP BY EmployeeNo
        HAVING Count(ProjectNO) > 2);
```

(c) Find the name of the employee who earns more than the average salary of all the employees.

```
SELECT EmployeeNo, EmployeeName, EmployeeBasicPay
FROM EMPLOYEE
WHERE EmployeeBasicPay >
        (SELECT avg (e.EmployeeBasicPay)
         FROM EMPLOYEE e);
```

3) (i) To find names of employees who are working on all projects.
(ii) To find names of the projects on which no employee is presently working.

Check Your Progress 3

1.

```
CREATE VIEW PROJECTDETAILS AS
SELECT p.ProjectNo, ProjectName, EmployeeName, EmpRoleInProject
FROM EMPLOYEE e, EMPLOYEEPROJECT ep, PROJECT p
WHERE e.EmployeeNo = ep.EmployeeNo  AND ep.ProjectNo = p.ProjectNo;
```

2.

```
CREATE SEQUENCE ODDSEQ3
START WITH 101
INCREMENT BY 2
 MAX VALUE 999;
```

3. Indexes are very useful for answering queries using non-key attributes. However, they enhance overheads in terms of maintaining indexes.

UNIT 9 ADVANCED SQL

Structure

9.0 INTRODUCTION

The Structured Query Language (SQL) is a standard query language for database systems. It is considered one of the factors contributing to the success of commercial database management systems, primarily because of the availability of this standard language on most commercial database systems. We have already discussed about SQL in the previous two Units, where we discussed the SQL commands for data definition, data manipulation, data control, joins, sub-queries, etc.

In this unit, we provide details of some of the advanced features of Structured Query Language. We will discuss Assertions and Views, Triggers, Standard Procedures and Cursors. The concepts of embedded and dynamic SQL and SQLJ, which are used along with JAVA, have also been introduced. Some of the advanced features of SQL have been covered. We will provide examples in various sections rather than including a separate section of examples. The examples given here are closer to SQL3 standard yet may not be applicable to every commercial database management system. For more details, you may go through the documentation of DBMS that you may use for implementation of database.

9.1 OBJECTIVES

After going through this unit, you should be able to:

- define Assertions and explain how they can be used in SQL;
- explain the concept of views, SQL commands on views and updates on views;
- define and use Cursors;
- discuss Triggers and write stored procedures; and
- explain Dynamic SQL and SQLJ.

9.2 ASSERTIONS AND VIEWS

One of the major requirements in a database system is to define constraints on various tables. Some of these simple constraints can be specified as primary key, NOT

NULL, check value and range constraints. Such constraints can be specified while creating the table using the statement CREATE TABLE using the clauses like NOT NULL, PRIMARY KEY, UNIQUE, CHECK etc. Referential constraints can be specified with the help of foreign key constraints. However, there are some constraints, which may relate to more than one table. These are called assertions. In this section, we will discuss two important concepts that we use in database systems, viz., views and assertions. Assertions are general constraints, while views are virtual tables. Let us discuss them in more detail.

9.2.1 Assertions

Assertions are general constraints. For example, a hypothetical University does not admit students whose age is more than 25 years OR is more than the minimum age of the teacher at that University. Such general constraints can be implemented with the help of an assertion statement. The syntax of an assertion statement is given below:
Syntax:

```
CREATE ASSERTION <Name>
CHECK (<Condition>);
```

The assertion name helps in identifying the constraints specified by the assertion. These names can be used to modify or delete an assertion later. Assuming that the University has a STUDENT and one FACULTY table, the assertion on the age of students can be implemented as:

```
CREATE ASSERTION age-constraint
CHECK (NOT EXISTS (
        SELECT *
        FROM STUDENT s
        WHERE s.age > 25
        OR s.age > (
                SELECT MIN (f.age)
                FROM FACULTY f
        )));
```

An assertion, as above, is enforced on a database system by the database management system such that the constraints stated in the assertion are not violated. Assertions are checked whenever a related relation changes.

Example 1: Consider the following relations of a university:
FACULTY (code, name, age, basic_salary, medical_allowance, other_benefits)
MEDICAL_CLAIM (code, claimdate, amount, comment)

Write an assertion for the University, which verifies that the total medical claims made by a faculty member in the financial year 2022-23 should not exceed his/her medical allowance.

```
CREATE ASSERTION Total_medical_claim
CHECK (NOT EXISTS (
        SELECT code, SUM (amount), MIN (medical_allowance)
        FROM (FACULTY NATRUAL JOIN MEDICAL_CLAIM)
        WHERE claimdate >= “01-04-2022” AND claimdate < “01-04-2023”
        GROUP BY code
        HAVING MIN(medical_allowance) < SUM(amount)
));
```

OR

```
CREATE ASSERTION Total_medical_claim
CHECK (NOT EXISTS (
        SELECT  *
```

```
FROM FACULTY f
WHERE f.code IN
        (SELECT code, SUM(amount)
         FROM MEDICAL_CLAIM m
         WHERE claimdate >="01-04-2022" AND claimdate < "01-04-2023"
                AND f.code=m.code
                AND  f.medical-allowance<SUM(amount)
        )
));
```

Please analyse both the queries above. So, now you can create an assertion. But how can these assertions be used in database systems? The general constraints may be designed as assertions, which can be put into the stored procedures. Thus, any violation of an assertion may be detected.

9.2.2 Views

A view is a virtual table, which does not actually store data. Then what does it contain? A view is a query on the physical tables that store the data. The SQL command for creating views is explained with the help of an example.

Example 2: A student's database has the following tables:

```
STUDENT (name, enrolmentno, dateofbirth)
MARKS  (enrolmentno, subjectcode, smarks)
```

For the database above a view can be created for a teacher, who is allowed to view only the performance of the student in his/her subject, let us say MCS207.

```
CREATE VIEW SUBJECT_PERFORMANCE AS

    (SELECT  s.enrolmentno, name, subjectcode, smarks
    FROM STUDENT s, MARKS m
    WHERE s.enrolmentno = m.enrolmentno AND
            subjectcode 'MCS207'
    ORDER BY s.enrolmentno;
```

A view can be dropped using a DROP statement as:

```
DROP VIEW SUBJECT_PERFORMANCE;
```

The physical table, which stores the data on which the statement of the view is written, is referred to as the base table. You can create views on two or more base tables by combining the data using JOIN. Thus, a view hides the logic of joining the tables from a user. You can also index the views to speed up the performance of query evaluation. Once a view has been created, it can be queried exactly like a base table. For example:

```
SELECT   *
FROM  STUDENT_PERFORMANCE
WHERE   smarks  > 50;
```

How are the views implemented?

There are two strategies for implementing the views. These are:

- Query modification
- View materialisation.

In the query modification strategy, any query that is made on the view is modified to include the view-defining expression. For example, consider the view

STUDENT_PERFORMANCE. Consider a query on this view as: "The teacher of the course MCS207 wants to find the maximum and average marks in the course." The query for this in SQL would be:

```
SELECT  MAX(smarks), AVG(smarks)
FROM SUBJECT_PERFORMANCE
```

Since SUBJECT_PERFORMANCE is itself a view the query will be modified automatically as:

```
SELECT MAX (smarks), AVG (smarks)
FROM STUDENT s, MARKS m
WHERE s.enrolmentno=m.enrolmentno AND subjectcode= "MCS207";
```

However, this approach has a major disadvantage. Consider that you are repeatedly executing a complex query on a view in a large database system, the query modification will have to be performed for every execution of the query leading to inefficient utilisation of resources such as time and space.

The view materialisation strategy solves this problem by creating a temporary physical table for a view. However, this strategy is not useful in situations where database tables are frequently updated, as it will require frequent updating of the materialised views.

Can views be used for Data Manipulations?

Some views can be used during DML operations like INSERT, DELETE and UPDATE. When you perform DML operations on a view, such modifications are passed to the underlying base tables. However, not all views allow DML operations. Conditions for the view that may allow Data Manipulation are:

A view allows data updating if it fulfils the following conditions:

1) View Created from a single table:
   - The view allows the INSERT operation if the PRIMARY KEY column(s) and all the NOT NULL columns are included in the view.
   - The view should not be defined using any aggregate function or GROUP BY, HAVING or DISTINCT clauses. This is because any update on aggregated attributes or groups cannot be traced back to a single tuple of the base table. For example, consider a view AVGMARKS (coursecode, avgmark) created on a base table STUDENT (st_id, coursecode, marks). In the AVGMARKS view, changing the class average marks for coursecode "MCS207" to 50 from a calculated value of 40, cannot be accounted for a single tuple in the STUDENT base table, as the average marks are computed from the marks of all the Student tuples for that coursecode. Thus, this update will be rejected.
2) The views in SQL that are defined using joins are normally NOT updatable in general.
3) WITH CHECK OPTION clause of SQL checks the updatability of data from views, therefore, must be used with views through which you want to update.

Views and Security

Views are useful for the security of data. A view allows a user to use the data that is available through the view; thus, the hidden data is not made accessible. Access privileges can be given on views. Let us explain this with the help of an example. Consider the view that we have created for teacher: STUDENT_PERFORMANCE. You can grant privileges to a teacher whose name is 'ABC' as:

GRANT SELECT, INSERT, DELETE ON STUDENT_PERFORMANCE TO ABC
WITH GRANT OPTION;

Please note that the teacher ABC has been given the rights to query, insert and delete the records on the given view. Please also note s/he is authorised to grant these access rights (WITH GRANT OPTION) to any other user. The access rights can be revoked using the REVOKE statement:

REVOKE ALL ON STUDENT_PERFORMANCE FROM ABC;

☞ Check Your Progress 1

1) Consider a constraint – the value of the age field of the student at a formal University should be between 17 years and 50 years. Would you like to write an assertion for this statement?

……

……

……

2) Create a view for finding the average marks of the students in various subjects for the tables given in example 2.

……

……

……

3) Can the view created in problem 2 be used to update subjectcode?

……

……

……

9.3 EMBEDDED SQL AND DYNAMIC SQL

SQL commands can be entered through a standard SQL command level user interface. Such interfaces are interactive in nature and the result of a command is shown immediately. Such interfaces are very useful for those who have some knowledge of SQL and want to create a new type of query. However, in a database application where a naïve user wants to make standard queries, that too using GUI like interfaces, probably an application program needs to be developed. Such interfaces sometimes require the support of a programming language environment.

Please note that SQL normally does not support a full programming paradigm (although the latest SQL has full API support), which allows it a full programming interface. In fact, most of the application programs are seen through a programming interface, where SQL commands are put wherever database interactions are needed. Thus, SQL is embedded into programming languages like C, C++, JAVA, etc. Let us discuss the different forms of embedded SQL in more detail.

9.3.1 Embedded SQL

The embedded SQL statements can be put in the application program written in C++, Java or any other host language. These statements sometime may be called static. Why are they called static? The term ‘static’ is used to indicate that the embedded SQL commands, which are written in the host program, do not change automatically during the lifetime of the program. Thus, such queries are determined at the time of database application design. For example, a *query statement* embedded in C++ to determine the status of a booking of a ticket for a train will not change. However, this

query may be executed for many different tickets. Please note that it will only change the input parameters to the query, which are ticket number, train number, date of boarding, etc., and not the query itself.

But how is such embedding done? Let us explain this with the help of an example.

Example 3: Write a C program segment that prints the details of a student whose enrolment number is given as input.

Let us assume the relation:

STUDENT (enrolno:char(9), name:Char(25), phone:integer(12), prog_code:char(3))

The C Program may be as follows:

```
/* add proper include statements*/
/*declaration in C program */
EXEC SQL BEGIN DECLARE SECTION;
        Char enrolno[10], name[26], p_code[4];
        int phone;
        int SQLCODE;
        char SQLSTATE[6]
EXEC SQL END DECLARE SECTION;

/* The connection needs to be established with SQL*/
/* Write appropriate connection statements */

/* program segment for the required function */
printf ("enter the enrolment number of the student");
scanf ("% s", &enrolno);
EXEC SQL
        SELECT  name,  phone,  prog_code INTO
                :name, :phone, :p_code
        FROM STUDENT
        WHERE  enrolno = :enrolno;
If (SQLCODE == 0)
            printf  ("%d,  %s, %s, %s", enrolno, name, phone, p-code)
        else
            printf ("Wrong Enrolment Number");
```

Please note the following points in the program above:

- The program is written in the host language 'C' and contains embedded SQL statements.
- Although in the program an SQL query (SELECT) has been added. You can embed any DML, DDL or view statements.
- The distinction between an SQL statement and a host language statement is made by using the keyword EXEC SQL; thus, this keyword helps in identifying the Embedded SQL statements.
- Please note that the statements including (EXEC SQL) are terminated by a semi-colon (;).
- As the data is to be exchanged between a host language and a database, there is a need for shared variables that are shared between the environments. Please note that enrolno[10], name[20], p_code[4] etc. are shared variables and declared in 'C'. The colon (:) character in the SQL statement precedes the shared variables to distinguish them from the Table attributes.
- Please note that the shared host variables enrolno is declared to have char[10] whereas, an SQL attribute enrolno has only char[9]. Why? Because in 'C' conversion to a string includes a '\ 0' as the end of the string.

- The type mapping between 'C' and SQL types is defined in the following table:

'C' TYPE	SQL TYPE
long	INTEGER
short	SMALLINT
float	REAL
double	DOUBLE
char [i+1]	CHAR (i)

- Please also note that these shared variables are used in the SQL statements of the program. They are prefixed with the colon (:) to distinguish them from database attribute and relation names. However, they are used without this prefix in any C language statement.
- Please also note that these shared variables have almost the same name (except p_code) as that of the attribute name of the database. The prefix colon (:) distinguishes whether we are referring to the shared host variable or an SQL attribute. Such similar names are a good programming convention as it helps in identifying the related attribute easily.
- Please note that the shared variables are declared between BEGIN DECLARE SECTION and END DECLARE SECTION and their type is defined in 'C' language.

Two more shared variables have been declared in 'C'. These are:

- SQLCODE as int.
- SQLSTATE as char of size 6.
- These variables are used to communicate errors and exception conditions between the database and the host language program. The value 0 in SQLCODE means successful execution of SQL command. A value of the SQLCODE =100 means 'no more data'. SQLCODE < 0 indicates an error. Similarly, SQLSTATE is a 5-char code the 6th character is for '\0' in the host language 'C'. Value "00000" in an SQLSTATE indicates no error. You can refer to DBMS/SQL standard in more detail for more information.
- In order to execute the required SQL command, connection with the database server need to be established by the program. For this, the following SQL statement is used:

  CONNECT <name of the server> AS <name of the connection>
  AUTHORISATION <username, password>;

  To disconnect, you can simply give the command:

  DISCONNECT <name of the connection>;

You must check all these statements from the documentation of the commercial database management system, which you are using.

Explanation of SQL query in the given program: In the given SQL query, first, the given value of the enrolment number is transferred to the SQL attribute value, the query then is executed and the result, which may be a single tuple in this case, is transferred to shared host variables as indicated by the keyword INTO after the SELECT statement.

The SQL query runs as a standard SQL query except for the use of shared host variables. The rest of the C program has very simple logic and will print the data of the students whose enrolment number has been entered.

Please note that in this query, as the enrolment number is the primary key to the relation, only one tuple will be transferred to the shared host variables. But what will

happen if the execution of embedded SQL query results in more than one tuple? Such situations are handled with the help of a cursor. Let us discuss such queries in more detail.

9.3.2 Cursors and Embedded SQL

Let us first define the term 'cursor'. The database server may allocate a portion of RAM for database interaction and internal processing. This portion may be used for query processing using SQL. This portion of RAM is also called the cursor. What should be the size of memory for the query processing? Ideally, the size of memory allotted for the query processing should be equal to the memory required to hold the query result. However, the available memory puts a constraint on the allotted size. Whenever a query results in several tuples, you can use a cursor to process the currently available tuples one by one. How? Let us explain the use of the cursor with the help of an example:

Since most of the commercial RDBMS architectures are client-server architectures, on the execution of an embedded SQL query, the resulting tuples are cached in the cursor. This operation is performed on the server. Sometimes, the cursor is opened by RDBMS itself – these are called **implicit** cursors. However, in embedded SQL you need to declare these cursors explicitly – these are called **explicit cursors.** Any cursor needs to have the following operations defined on them:

DECLARE – to declare the cursor.
OPEN AND CLOSE - to open and close the cursor.
FETCH – get the current records one by one till the end of tuples.

In addition, cursors have some attributes through which we can determine the state of cursor. These may be:

ISOPEN – It is true if the cursor is OPEN, otherwise false.
FOUND/NOT FOUND – It is true if a row is fetched successfully/not successfully.
ROWCOUNT – It determines the number of tuples in the cursor.

Let us explain the use of the cursor with the help of an example:

Example 4: Write a C program segment that inputs the final grade of the students of PGDCA programme.

Let us assume the relation:

STUDENT (enrolno:char(9), name:Char(25), phone:integer(12),
prog_code:char(3)); grade: char(1));

The program segment is:

```
/* add proper include statements*/

/*declaration in C program */
EXEC SQL BEGIN DECLARE SECTION;
        Char enrolno[10], name[26], p_code[4], grade; /* grade is just one character*/
        int phone;
        int SQLCODE;
        char SQLSTATE[6]
EXEC SQL END DECLARE SECTION;

/* The connection needs to be established with SQL*/
/* Write appropriate connection statements */

/* program segment for the required function */
```

```
printf ("enter the programme code);
scanf ("%s, &p_code);
EXEC SQL DECLARE CURSOR FINALGRADE
        SELECT enrolno, name, phone, grade
        FROM STUDENT
        WHERE prog_code =:p_code
        FOR UPDATE OF grade;
EXEC SQL OPEN FINALGRADE;
EXEC SQL FETCH FROM FINALGRADE
        INTO :enrolno, :name, :phone, :grade;
WHILE (SQLCODE==0) {
        printf ("enter grade for enrolment number, "%s", enrolno);
        scanf ("%c", grade);
        EXEC SQL
                UPDATE STUDENT
                SET grade=:grade
                WHERE CURRENT OF FINALGRADE
        EXEC SQL FETCH FROM FINALGRADE
                INTO :enrolno, :name, :phone, :grade;
}
EXEC SQL CLOSE FINALGRADE;
```

- Please note that the declared section remains almost the same. The cursor is declared to contain the output of the SQL statement. Please notice that in this case, there will be many tuples of students database, which belong to a particular programme.
- The purpose of the cursor is also indicated during the declaration of the cursor.
- The cursor is then opened and the first tuple is fetched into the shared host variable followed by an SQL query to update the required record. Please note the use of CURRENT OF which states that these updates are for the current tuple referred to by the cursor.
- WHILE Loop is checking the SQLCODE to ascertain whether more tuples are pending in the cursor.
- Please note that the SQLCODE will be set by the last fetch statement executed just prior to the WHILE condition check.

How are these SQL statements compiled and error checked during embedded SQL?

The SQL pre-compiler performs the type checking of the various shared host variables to find any mismatches or errors on each SQL statement. It then stores the results in the SQLCODE or SQLSTATE variables.

Is there any limitation on these statically embedded SQL statements?

They offer only limited functionality, as the query must be known at the time of application development so that they can be pre-compiled in advance. However, many queries are not known at the time of development of an application; thus, you require dynamically embedded SQL that are discussed next.

9.3.3 Dynamic SQL

Dynamic SQL, unlike embedded SQL statements, is built at the run time and placed in a string in a host variable. The created SQL statements are then sent to the DBMS for processing. Dynamic SQL is generally slower than statically embedded SQL as they require complete processing including access plan generation during the run time.

However, they are more powerful than *embedded SQL* as they allow run-time application logic. The basic advantage of using dynamic embedded SQL is that you

need not compile and test a new program for a new query. Let us explain the use of dynamic SQL with the help of an example.

Example 5: Write a dynamic SQL interface that allows a student to get and modify permissible details about him/her. The student may ask for a subset of information also. Assume that the student database has the following relations.

STUDENT (enrolno, name, dob)

RESULT (enrolno, coursecode, marks)

In the table above, a student has access rights for accessing information on his/her enrolment number, but s/he cannot update the data. Assume that the username used by a student is his/her enrolment number.

A sample program segment is given below (please note that the syntax may change for different commercial DBMSs).

```
/* declarations in SQL */
EXEC SQL BEGIN DECLARE SECTION;
        char inputfields (50);
        char tablename(10)
        char sqlquerystring(200)
EXEC SQL END DECLARE SECTION;

        printf  ("Enter the fields you want to see \n");
        scanf ("%s", &inputfields);
        printf ("Enter the name of table STUDENT or RESULT");
        scanf ("%s", &tablename);
        sqlqueryystring = "SELECT" +inputfields +" "+
                "FROM" + tablename
                + "WHERE enrolno + :USER"
/*Plus is used as a symbol for concatenation operator; in some DBMS it may be ||*/
/* Assumption: the username is available in the host language variable USER*/

EXEC SQL PREPARE sqlcommand FROM :sqlqueryystring;
EXEC SQL EXECUTE sqlcommand;
```

Please note the following points in the example above.

- The query can be entered completely as a string by the user.
- The query can be fabricated using a concatenation of strings. The program segment is language dependent and is not portable to other programming languages or environments.
- Query modification may be performed, keeping data security in mind.
- The query is prepared and executed using a suitable SQL EXEC command.

9.3.4 SQLJ

Till now we have discussed embedding SQL in C, can you embed SQL statements into JAVA Program? For inserting SQL statements in JAVA, you use SQLJ. In SQLJ, a preprocessor called SQLJ translator translates the SQLJ source file to a JAVA source file. The JAVA file is compiled and run on the database. The use of SQLJ improves the productivity and manageability of JAVA Code as:

- The code becomes somewhat compact.
- No run-time SQL syntax errors as SQL statements are checked at compile time.
- It allows sharing of JAVA variables with SQL statements. Such sharing is not possible otherwise.

SQLJ provides a standard form in which SQL statements can be embedded in the JAVA program. SQLJ statements always begin with a #sql keyword. These embedded SQL statements are of two categories – Declarations and Executable Statements.

Declarations have the following syntax:

```
#sql <modifier> context context_classname;
```

The executable statements have the following syntax:

```
#sql {SQL operation returning no output};

        OR

#sql result = {SQL operation returning output};
```

Example 6: Write a JAVA function to print the student details of the student table, for the students who have been admitted in 2023 or later and whose names are like 'As'.

Assuming that the first two digits of the 9-digit enrolment number represent the year of admission, the required input conditions may be:

- The enrolment number should be more than "23000000" and
- The name contains the substring "As".

Please note that these input conditions will not be part of the Student Display function, rather will be used in the main ( ) function that may be created separately by you. The following display function will accept the values as the parameters supplied by the main ( ).

```
public void DISPSTUDENT (String enrolno, String name, int phone)
{
    try {
                SelRowIter      srows = null;
               # sql srows = {  SELECT enrolno, name, phone
                               FROM STUDENT
                          WHERE   enrolno > 230000000 AND name LIKE "As%"
                               };
                while ( srows.next ( ) ) {
                      int enrolno   =  srows.enrolno ( );
                      String name =  srows.name ( );
                      int phone     =  srows.phone ( );
                      System.out.println ( "Enrollment_No  =  " + enrolno);
                      System.out.println ("Name =" +name);
                      System.out.println ("phone =" +phone);
                }
    } Catch (Exception e) {
            System.out.println ( " error accessing database" + e.toString());
    }
}
```

☞ Check Your Progress 2

1) A University decided to enhance the marks of all the students in the subject MCS207 by 2. (Table to be used: RESULT (enrolno, coursecode, marks)). Write a segment of the embedded SQL program to do this processing.

……

……

……

2) What is dynamic SQL?

……

……

……..

3) Can SQL be embedded in JAVA?

……

……

……

9.4 STORED PROCEDURES AND TRIGGERS

In this section, we will discuss some standard features that make commercial databases a strong implementation tool for information systems. These features are triggers and stored procedures. Let us discuss them in more detail in this section.

9.4.1 Stored Procedures

Stored procedures are collections of small programs that are stored in compiled form. A stored procedure is designed for the implementation of a specific function. For example, a company may have rules such as:

- A code (like an enrolment number) contains a check digit to check the validity of the code.
- The date of change of any value may be recorded.

These rules may apply to every application that uses that code. Thus, instead of inserting them in the code of each application they may be put in a stored procedure and reused.

The use of procedure has the following advantages from the viewpoint of database application development.

- They help in removing SQL statements from the application program, thus making the program more readable and maintainable.
- They run faster than SQL statements since they are already compiled in the database.

Stored procedures can be created using CREATE PROCEDURE statement in some commercial DBMS.

Syntax:

```
CREATE [or replace] PROCEDURE [user] <ProcedureName>
                [(argument datatype [, argument datatype]….)]
BEGIN
        Host Language statements;
END;
```

Example 7: Consider a data entry application that allows entry of the enrolment number, first name and last name of a student. The application then combines the first and last name and enters the complete name in uppercase in the table STUDENT. The student table has the following structure:

STUDENT (enrolno:char(9), name:char(40));

The stored procedure for this application may be written (using an embedded programming language) as:

```
CREATE PROCEDURE studententry (
        enrolment   INOUT  char (9);
        f_name       INOUT  char (20);
        l_name       INOUT  char (20);
BEGIN
 /* change all the characters to uppercase and trim the length */
        f_name = TRIM(UPPER(f_name));
        l_name  = TRIM(UPPER (l_name));
        name   =  f_name || " " || l_name;
        INSERT INTO CUSTOMER
                VALUES (enrolment, name);
END;
```

INOUT used in the host language indicates that this parameter may be used both for the input and output of values in the database.

While creating a procedure, if you encounter errors, then you can use the ***show errors*** command. It shows all the errors encountered by the most recently created procedure object.

You can also write an SQL command to display errors. The syntax for finding an error in a commercial database is:

```
SELECT  *
FROM USER_ERRORS
WHERE name ='procedure name' and type='PROCEDURE';
```

Procedures are compiled by the DBMS. However, if there is a change in the tables then the procedure needs to be recompiled. You can recompile the procedure explicitly using the following command:

```
ALTER PROCEDURE procedure_name COMPILE;
```

You can drop a procedure by using DROP PROCEDURE command.

9.4.2 Triggers

Triggers are somewhat like stored procedures except that they are activated automatically. When is a trigger activated? A trigger is activated on the occurrence of a particular event. What are these events that can cause the activation of triggers? These events may be database update operations like INSERT, UPDATE, DELETE etc. A trigger consists of these essential components:

- An event that causes its automatic activation.
- The condition that determines whether the event has called an exception.
- The action that is to be performed.

Triggers do not take parameters and are activated automatically, thus, are different to stored procedures on these accounts. Triggers are used to implement constraints among more than one table. Specifically, the triggers should be used to implement the constraints that are not implementable using referential integrity or constraints. An instance of such a situation may be when an update in one relation affects a few tuples in another relation. However, please note that you should not be over-enthusiastic for

writing triggers – if any constraint is implementable using declarative constraints such as PRIMARY KEY, UNIQUE, NOT NULL, CHECK, FOREIGN KEY, etc., then it should be implemented using those declarative constraints rather than triggers, primarily due to performance reasons.

You may write triggers that may execute once for each row in a transaction – called Row Level Triggers, or once for the entire transaction - called Statement Level Triggers. Remember that you must have proper access rights on the tables on which you are creating the trigger. The following is the syntax of triggers in one of the commercial DBMSs:

```
CREATE  TRIGGER <trigger_name>
[BEFORE | AFTER|
<Event>
ON <tablename>
[WHEN <condition> | FOR EACH ROW]

<Declarations of variables, if needed, – may be used when creating trigger
using host language>

BEGIN
        <SQL statements OR host language SQL statements>

[EXCEPTION]

        <Exceptions if any>
END;
```

Let us explain the use of triggers with the help of an example:

Example 8: Consider the following relation of a student's database:

STUDENT(enrolno, name, phone)
RESULT (enrolno, coursecode, marks)
COURSE (course-code, c-name, details)

Assume that the marks are out of 100 in each course. The passing marks in a subject are 50. The University has a provision for 2% grace marks for the students who are failing marginally – that is, if a student has 48 marks, s/he is given 2 marks grace and if a student has 49 marks, then s/he is given 1 grace mark. Write the suitable trigger for this situation.

Please note the requirements of the trigger:

Event: UPDATE of marks
OR
INSERT of marks
Condition: When a student has 48 OR 49 marks

Action: Give 2 grace marks to the student having 48 marks and 1 grace mark to the student having 49 marks.

The trigger for this thus can be written as:

```
CREATE TRIGGER grace
AFTER INSERT OR UPDATE OF marks ON RESULT
WHEN (marks = 48 OR marks =49)
UPDATE RESULT
        SET marks =50;
```

We can drop a trigger using a DROP TRIGGER statement.

```
DROP TRIGGER trigger_name;
```

The triggers are implemented in many commercial DBMSs. Please refer to the documentation of the respective DBMS for more details.

9.5 ADVANCED FEATURES OF SQL

The latest SQL standards have enhanced SQL tremendously. Let us touch upon some of these enhanced features. More details on these would be available in the online sources of SQL.

SQL Interfaces: SQL also has good programming level interfaces. The SQL supports a library of functions for accessing a database. These functions are also called the Application Programming Interface (API) of SQL. The advantage of using an API is that it provides flexibility in accessing multiple databases in the same program irrespective of DBMS, while the disadvantage is that it requires more complex programming. The following are two common functions, called interfaces:

SQL/CLI (SQL – call level interface) is an advanced form of Open Database Connectivity (ODBC).

Java Database Connectivity (JDBC) – allows object-oriented JAVA to connect to multiple databases.

SQL support for object orientation: The latest SQL standard also supports object-oriented features. It allows the creation of abstract data types, nested relations, object identifiers etc.

Interaction with Newer Technologies: SQL provides support for XML (eXtended Markup Language) and Online Analytical Processing (OLAP) for data warehousing technologies.

☞ Check Your Progress 3

1) What is the stored procedure?

……

……

……

2) Write a trigger that restricts updating of the STUDENT table outside the normal working hours/holiday.

……

……

……

9.6 SUMMARY

This unit has introduced some important advanced features of SQL. The unit has also provided information on how to use these features.

An assertion can be defined as the general constraint on the state of a database. These constraints are expected to be always satisfied by the database. The assertions can be stored as stored procedures.

Views are the external schema windows of the data from a database. Views can be defined on a single table or multiple tables and help in automatic security of hidden

data. All the views cannot be used for updating data in the tables. You can query a view.

The embedded SQL helps in providing complete host language support to the functionality of SQL, thus making application programming somewhat easier. An embedded query can result in a single tuple, however, if it results in multiple tuples then you need to use cursors to perform the desired operation. Cursor is a sort of pointer to the area in the memory that contains the result of a query. The cursor allows sequential processing of these result tuples. The SQLJ is the embedded SQL for JAVA. Dynamic SQL is a way of creating queries through an application and compiling and executing them at the run time. Thus, it provides a dynamic interface and hence the name.

Stored procedures are precompiled procedures and can be invoked from application programs. Arguments can be passed to a stored procedure and can return values. A trigger is also like a stored procedure except that it is invoked automatically on the occurrence of a specified event.

You should refer to the further readings for more details on the topics relating to this unit.

9.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) The constraint is a simple Range constraint and can easily be implemented with the help of declarative constraint statement. (You need to use CHECK CONSTRAINT statement). Therefore, there is no need to write an assertion for this constraint.

2)
```
CREATE VIEW avgmarks AS (
   SELECT subjectcode, AVG(smarks)
   FROM MARKS
   GROUP BY subjectcode);
```

3) No, the view is using a GROUP BY clause. Thus, if you try to update the subjectcode, you cannot trace it back to a single tuple where such a change needs to take place.

Check Your Progress 2

1)
```
/*The table is RESULT (enrolno, coursecode, marks). */
EXEC SQL BEGIN DECLARE SECTION;
    char enrolno [10], coursecode[7];
    int marks;
    int SQLCODE;
    char SQLSTATE[6]
EXEC SQL END DECLARE SECTION;
/*The connection needs to be established with SQL*/
/* program segment for the required function*/
    printf ("enter the course code for which 2 marks are to be added");
    scanf ("%s", &coursecode);
EXEC SQL DECLARE CURSOR GRACE
  SELECT enrolno, coursecode, marks
  FROM RESULT
  WHERE coursecode=:coursecode
```

```
/* For update of marks */
EXEC SQL OPEN GRACE;
EXEC SQL FETCH FROM GRACE
        INTO :enrolno, :coursecode, :marks;
WHILE (SQL CODE==0)
    {
    EXEC SQL
        UPDATE RESULT
        SET marks = marks+2
        WHERE CURRENT OF GRACE;
    EXEC SQL FETCH FROM GRACE
        INTO :enrolno, :coursecode, :marks;
    }
    EXEC SQL CLOSE GRACE;
```

An alternative implementation in a commercial database management system may be:

```
DECLARE CURSOR grace IS
    SELECT enrolno, coursecode, marks
    FROM RESULT
    WHERE coursecode ='MCS207';
    str_enrolno RESULT.enrolno%type;
    str_coursecode RESULT.coursecode%type;
    str_marks RESULT.marks%type;
BEGIN
    OPEN grace;
    IF GRACE %OPEN THEN
       LOOP
        FETCH grace INTO str_enrolno, str_coursecode, str_marks;
        Exit when grace%NOTFOUND;
            UPDATE student SET marks=str_marks +2;
                INSERT INTO resultmcs207 VALUES
                (str_enrolno, str_coursecode, str_marks);
      END LOOP;
     COMMIT;
     CLOSE grace;
    ELSE
        Dbms_output.put_line ('Unable to open cursor');
    END IF;
END;
```

2) Dynamic SQL allows run-time query-making through embedded languages. The basic step here would be - create a valid query string and then execute that query string. Since the queries are compiled and executed at the run time thus, it is slower than embedded SQL.

3) Yes. You may use SQLJ for this purpose.

Check Your Progress 3

1) Stored procedure is a compiled procedure in a host language that has been written for a specific purpose.

2) The trigger is some pseudo-DBMS may be written as:

```
CREATE TRIGGER resupdstrudent
        BEFORE INSERT OR UPDATE OR DELETE ON STUDENT
BEGIN
```

```
        IF (DAY (SYSDATE) IN ('SAT', 'SUN')) OR
            (HOURS (SYSDATE) NOT BETWEEN '09:00' AND 18:30')
        THEN
            RAISE_EXCEPTION AND OUTPUT ('OPERATION NOT
                                ALLOWED AT THIS TIME/DAY');
        END IF;
    END;
```

Please note that we have used some hypothetical functions, the syntax of which will be different in different RDBMS.