UNIT 5 DATABASE INTEGRITY, FUNCITONAL DEPENDENCY AND NORMALISATION

Structure **Page Nos.**

5.0 INTRODUCTION

The first block of this course discusses about 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 normalize database.

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 attributes. 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 doing 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).

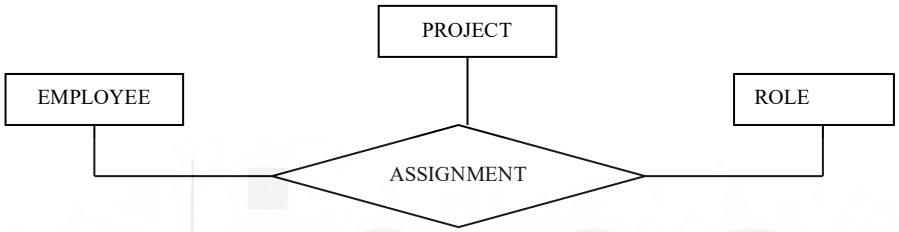


Figure 5.1: E-R diagram for employee role in development

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 employee 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

Modify or update operation 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

P

PNO	**PNAME**	**COLOUR**	**CITY**
P1	Nut	Red	Delhi
P2	Bolt	Blue	Pune

S3	Ballav	Pune
S4	Seema	Delhi
S5	Salim	Agra

P3	Part1	White	Mumbai
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 relation, 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 relation 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	Student Name	Student StAddress	Course Number	Course Name	Instructor	Office number of the Instructor
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 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 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 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 to update 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 as 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 CoNo. 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 number and name of a new course in the database, it would not be possible until a student enrols in that course. Similarly, information about a new student cannot be inserted in the database until the student enrols in a course. These problems are called insertion anomalies.

3. *Deletion Anomalies*: Loss of Useful Information: 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 StName "Rohan" and CoNo 'MCS-202" will result in loss of information that MCS-202 is named "Computer organization" 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*:

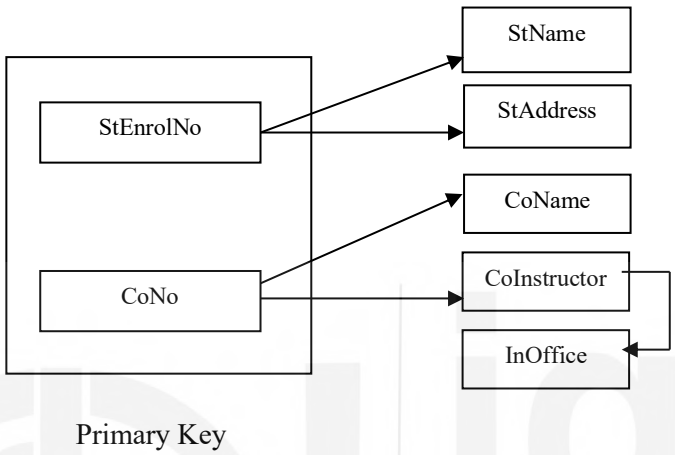


Figure 5.4: The dependencies of relation in Figure 5.3

In the *Figure 5.4*, an arrow shows dependence of a data attributes. For example, you may notice that two arrows are originating from an attribute student enrolment number (*StEnrolNo*). One of these arrows is to student name attribute (*StName*) and other 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 of 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 set 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 Y component is uniquely determined by the value of 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 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 the 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 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.

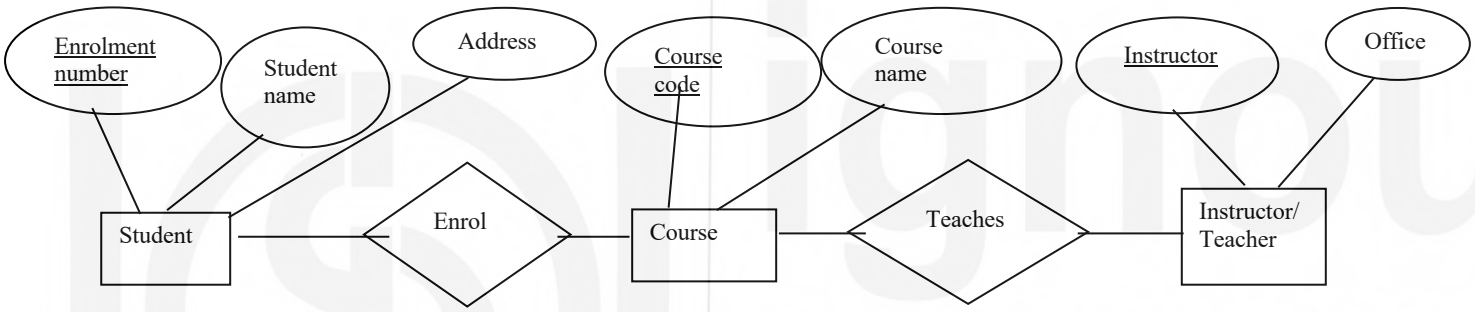


Figure 5.5: ERD of Entities – Student, Course and Teacher

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

Relations of Entities

STUDENT (enrno, StName, StAddress)

FDs: enrno → StName, StAddress

COURSE (CoCode, CoName)

FD: CoCode → CoName

INSTRUCTOR (inId, InOffice) // inID is instructor Id

FD: inId → InOffice

Relations of Relationships

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

FD: enrno, CoCode → 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 (CoCode, CoName, inId)

FDs: CoCode → CoName, inId

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 multivalue 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, minimizing redundancy, and
- secondly minimizing 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 normalization should include any or both of thr following properties.

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

Do you always normalize database to highest normalized 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 fulfils the following criteria (assuming the relation has only one candidate key, which is selected as the primary key of the relation):

(i) The relation fulfils 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 consist 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 forms 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 the 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 dependence 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 name of the instructor uniquely determines 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 using CoNo attribute. So, 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 exists

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 the 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.

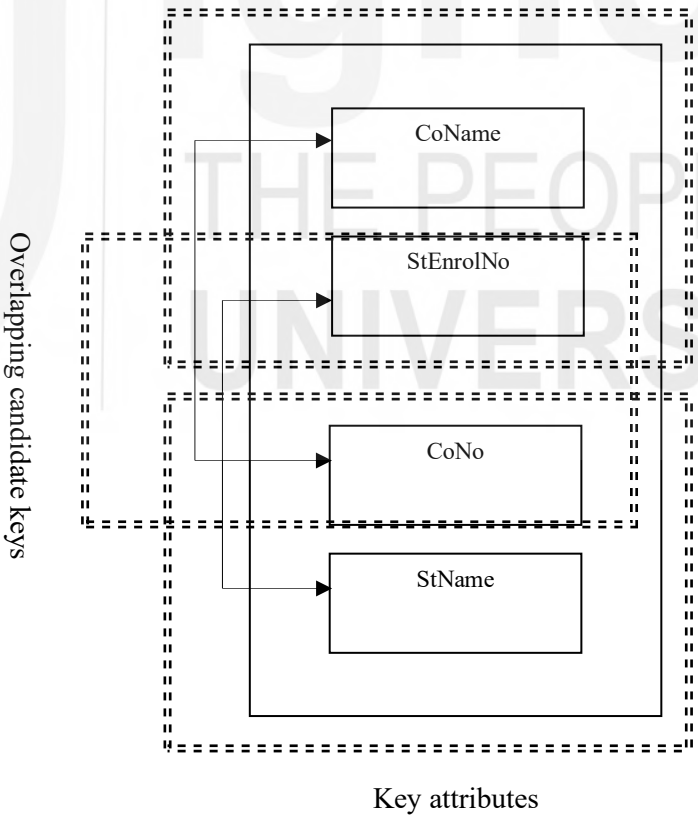


Figure 5.6: Functional Diagram for STUDENT5 relation

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 more 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 the 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 fulfils the following criteria:*

(i) The relation fulfils 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 key are functionally dependent.

For example, in the relation STUDENT5, the candidate keys:
(StEnrolNo, CoName) and (StName, CoName) has 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-ovelapping 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 checks 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 normalization 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 these projections are non-disjoint? The non-disjoint relations allow reconstruction of 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 fulfil 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 one basic point:

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

"Mormalisation 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 relation.
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 group 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 relation 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 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 second normal form. To achieve this, you isolate the attributes that depends on the entire composite key from the attributes those depends 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
103	3
103	4
104	5
104	3

DBMSlist	
DBMScode	**DBMS**
1	MySQL
2	DB2
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.

EMPLOYEEElist		
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 functional dependencies should 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 about different forms of the 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 the 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 key 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 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 book, which has not been issued to a student. This is 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 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.

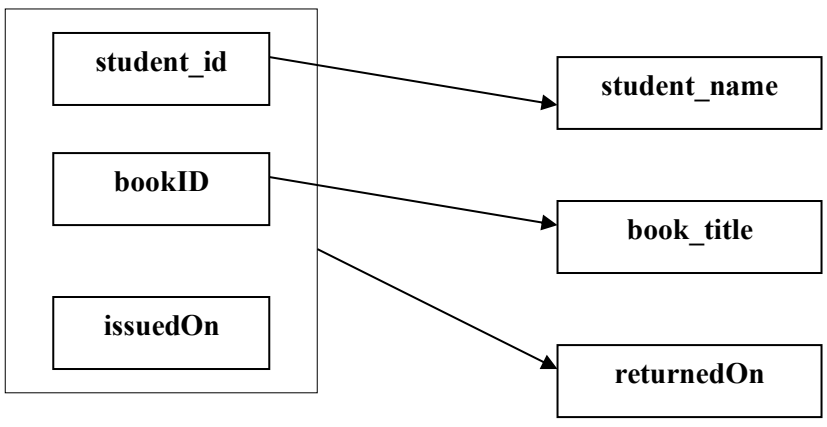
Just one candidate key, which is also chosen as the primary key, is : bookID, student_id, issuedOn.

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 is no overlapping candidate keys.

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

MEMBER

Member - ID	Member Name
A 101	Abhishek
R 102	Raman

BOOK

Book Codes	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 key.
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.

ignou
THE PEOPLE'S
UNIVERSITY

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 about 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 has been designed to address this problem.

This Unit focusses 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 a 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 upto 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 attribute 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 attributes. 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 contains more than one multivalued attributes. 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 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 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 employee number can determine all the tools in which employee is skillful. Further, both the *project* and *tool* attributes are independent of each other. Therefore, the following MVDs holds in the 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 the Figure 6.2, the two such tuples are:

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

Then two more tuples must exists as follows:

E001	DBMS	Virtualization
E001	Ecommerce	Python

Please check these two are Tuple 2 and Tuple 3 in the 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 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.

Trival 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 attributes, you should decompose it into 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 form of dependencies? Researcher has 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 the 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 minimize 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 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 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 type 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, 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 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 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 (<u>studentid, courseid</u>, grade) with the FD

studentid, coursed→grade

Represent this FD using template dependency.

Description	<u>studentid</u>	<u>courseid</u>	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 constrains 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:

Key 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 uphold 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 represents 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 digit 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 attributes, 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 holds 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 exists 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 of 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.

ignou
THE PEOPLE'S
UNIVERSITY

For Unit 7 & 8 : Please read the following unit of MCS-23 Block 2 Unit 1

UNIT 1 THE STRUCTURED QUERY LANGUAGE

Structure **Page Nos.**

1.0 INTRODUCTION

Database is an organised collection of information about an entity having controlled redundancy and serves multiple applications. DBMS (database management system) is an application software that is developed to create and manipulate the data in database. A query language can easily access a data in a database. SQL (Structured Query Language) is language used by most relational database systems. IBM developed the SQL language in mid-1979. All communication with the clients and the RDBMS, or between RDBMS is via SQL. Whether the client is a basic SQL engine or a disguised engine such as a GUI, report writer or one RDBMS talking to another, SQL statements pass from the client to the server. The server responds by processing the SQL and returning the results. The advantage of this approach is that the only network traffic is the initial query and the resulting response. The processing power of the client is reserved for running the application.

SQL is a data sub-language consisting of three built-in languages: Data definition language (DDL), Data manipulation language (DML) and Data control language (DCL). It is a fourth generation language. SQL has many more features and advantages. Let us discuss the SQL in more detail in this unit. It should be noted that many commercial DBMS may or may not implement all the details given in this unit. For example, MS-ACCESS does not support some of these features. Even some of the constructs may not be portable, please consult the DBMS Help for any such difference.

1.1 OBJECTIVES

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

- create, modify and delete database schema objects;
- update database using SQL command;
- retrieve data from the database through queries and sub-queries;
- handle join operations;
- control database access;
- deal with database objects like Tables, Views, Indexes, Sequences, and Synonyms using SQL.

1.2 WHAT IS SQL?

Structured Query Language (SQL) is a standard query language. It is commonly used with all relational databases for data definition and manipulation.

All the relational systems support SQL, thus allowing migration of database from one DBMS to another. In fact, this is one of the reasons of the major success of Relational DBMS. A user may be able to write a program using SQL for an application that involves data being stored in more than one DBMSs, provided these DBMS support standard SQL. This feature is commonly referred to as portability. However, not all the features of SQL implemented in one RDBMS are available in others because of customization of SQL. However, please note there is just ONE standard SQL.

SQL provides an interface where the user can specify "What" are the expected results. The query execution plan and optimisation is performed by the DBMS. The query plan and optimisation determines how a query needs to be executed. For example, if three tables X, Y and Z are to be joined together then which plan (X JOIN Y) and then Z or X JOIN (Y JOIN Z) may be executed. All these decisions are based on statistics of the relations and are beyond the scope of this unit.

SQL is called a non-procedural language as it just specifies what is to be dome rather than how it is to be done. Also, since SQL is a higher-level query language, it is closer to a language like English. Therefore, it is very user friendly.

The American National Standard Institute (ANSI) has designed standard versions of SQL. The first standard in this series was created in 1986. It was called SQL-86 or SQL1. This standard was revised and enhanced later and SQL-92 or SQL-2 was released. A newer standard of SQL is SQL3 which is also called SQL- 99. In this unit we will try to cover features from latest standards. However, some features may be found to be very specific to certain DBMSs.

Some of the important features of SQL are:

- It is a non procedural language.
- It is an English-like language.
- It can process a single record as well as sets of records at a time.
- It is different from a third generation language (C& COBOL). All SQL statements define what is to be done rather than how it is to be done.
- SQL is a data sub-language consisting of three built-in languages: Data definition language (DDL), Data manipulation language (DML) and Data control language (DCL).
- It insulates the user from the underlying structure and algorithm.
- SQL has facilities for defining database views, security, integrity constraints, transaction controls, etc.

There are many variants of SQL, but the standard SQL is the same for any DBMS environment. The following table shows the differences between SQL and one of its superset SQL*Plus which is used in Oracle. This will give you an idea of how various vendors have enhanced SQL to an environment. The non-standard features of SQL are not portable across databases, therefore, should not be used preferably while writing SQL queries.

Difference between SQL and SQL*Plus

SQL	SQL*Plus
SQL is a language	SQL *Plus is an environment
It is based on ANSI (American National Standards Institute) standard	It is oracle proprietary interface for executing SQL statements

SQL	
It is entered into the SQL buffer on one or more lines	It is entered one line at a time, not stored in the SQL buffer
SQL cannot be abbreviated	SQL*Plus can be abbreviated
It uses a termination character to execute command immediately	It does not require termination character. Commands are executed immediately.
SQL statements manipulate data and table definition in the database	It does not allow manipulation of values in the database

1.3 DATA DEFINITION LANGUAGE

As discussed in the previous block, the basic storage unit of a relational database management system is a table. It organises the data in the form of rows and columns. But what does the data field column of table store? How do you define it using SQL?

The Data definition language (DDL) defines a set of commands used in the creation and modification of schema objects such as tables, indexes, views etc. These commands provide the ability to create, alter and drop these objects. These commands are related to the management and administrations of the databases. Before and after each DDL statement, the current transactions are implicitly committed, that is changes made by these commands are permanently stored in the databases. Let us discuss these commands in more detail:

CREATE TABLE Command

Syntax:

```
CREATE TABLE <table name>(
Column_name1 data type (column width) [constraints],
Column_name2 data type (column width) [constraints],
Column_name3 data type (column width) [constraints],
……..
);
```

Where table name assigns the name of the table, column name defines the name of the field, data type specifies the data type for the field and column width allocates specified size to the field.

Guidelines for creation of table:

- Table name should start with an alphabet.
- In table name, blank spaces and single quotes are not allowed.
- Reserve words of that DBMS cannot be used as table name.
- Proper data types and size should be specified.
- Unique column name should be specified.

Column Constraints: NOT NULL, UNIQUE, PRIMARY KEY, CHECK, DEFAULT, REFERENCES,

On delete Cascade: Using this option whenever a parent row is deleted in a referenced table then all the corresponding child rows are deleted from the referencing table. This constraint is a form of referential integrity constraint.

Example 1:

```
CREATE TABLE product
    (
pno number (4) PRIMARY KEY,
pname char (20) NOT NULL,
qoh number (5) DEFAULT (100),
```

```
    class char (1) NOT NULL,
    rate number (8,2) NOT NULL,
     CHECK ((class='A' AND rate<1000) OR
    (class='B' AND rate>1000 AND rate<4500) OR
    (class='C' AND rate>4500))
);
```

The command above creates a table. Primary key constraint ensures that product number (pno) is not null and unique (both are the properties of primary key). Please note the use of data type char (20). In many implementations of SQL on commercial DBMS like SQL server and oracle, a data type called varchar and varchar2 is used respectively. Varchar basically is variable length character type subject to a maximum specified in the declarations. We will use them at most of the places later.

Please note the use of check constraints in the table created above. It correlates two different attribute values.

Example 2:

```
CREATE TABLE prodtrans
(
 pno number (4)
 ptype char (1) CHECK (ptype in ('I','R','S')),
 qty number (5)
FOREIGN KEY pno REFERENCES product (pno)
ON DELETE CASCADE);
```

In the table so created please note the referential constraint on the foreign key **pno** in **prodtrans** table to **product** table. Any product record if deleted from the product table will trigger deletion of all the related records (ON DELETE CASCADE) in the **prodtrans** table.

ALTER TABLE Command: This command is used for modification of existing structure of the table in the following situation:

- When a new column is to be added to the table structure.
- When the existing column definition has to be changed, i.e., changing the width of the data type or the data type itself.
- When integrity constraints have to be included or dropped.
- When a constraint has to be enabled or disabled.

Syntax

ALTER TABLE <table name> ADD (<column name> <data type>…);
ALTER TABLE <table name> MODIFY (<column name> <data type>…);
ALTER TABLE <table name> ADD CONSTRAINT <constraint name> < constraint type>(field name);
ALTER TABLE <table name> DROP<constraint name>;
ALTER TABLE <table name> ENABLE/DISABLE <constraint name>;

You need to put many constraints such that the database integrity is not compromised.

Example 3:

```
ALTER TABLE emp MODIFY (empno NUMBER (7));
```

DROP TABLE Command:

When an existing object is not required for further use, it is always better to eliminate it from the database. To delete the existing object from the database the following command is used.
Syntax:

DROP TABLE<table name>;

Example 4:

DROP TABLE emp;

1.4 DATA MANIPULATION LANGUAGE

Data manipulation language (DML) defines a set of commands that are used to query and modify data within existing schema objects. In this case commit is not implicit that is changes are not permanent till explicitly committed. DML statements consist of SELECT, INSERT, UPDATE and DELETE statements.

SELECT Statement

This statement is used for retrieving information from the databases. It can be coupled with many clauses. Let us discuss these clauses in more detail:

1. **Using Arithmetic operator**

Example 5:

```
SELECT ENAME, SAL, SAL+300
 FROM EMP;
```

2. **Operator Precedence**

The basic operators used in SQL are * / + -
Operators of the same priority are evaluated From Left to right
Parentheses are used to force prioritized evaluation.

Example 6:

```
SELECT ENAME, SAL, 12*SAL+100
FROM EMP;

SELECT ENAME, SAL, 12*(SAL+100)
FROM EMP;
```

3. **Using Column aliases**

Example 7:

To print column names as NAME and ANNUAL SALARY

```
SELECT ENAME "NAME", SAL*12 "ANNUAL SALARY"
FROM EMP;
```

4. **Concatenation operator**

Example 8:

Printing name and job as one string as column name employees:

```
SELECT ENAME||JOB "EMPLOYEES"
FROM EMP;
```

5. **Using Literal Character String**

Example 9:

To print <name> IS A <job> as one string with column name employee

```
SELECT ENAME || ' IS A ' || JOB AS "EMPLOYEE"
FROM EMP;
```

6. **To eliminate duplicate rows (distinct operator)**

Example 10:

```
SELECT DISTINCT DEPTNO
FROM EMP;
```

7. **Special comparison operator** used in where Clause

a. **between. …and…**It gives range between two values (inclusive)

Example 11:

```
SELECT ENAME, SAL
FROM EMP
WHERE SAL BETWEEN 1000 AND 1500;
```

b. **In (list):** match any of a list of values

Example 12:

```
SELECT EMPNO, ENAME, SAL, MGR
FROM EMP
WHERE MGR IN (7902, 7566, 7788);
```

7902, 7566, and 7788 are Employee numbers

c. **Like:** match a character pattern

- Like operator is used only with char and Varchar2 to match a pattern
- % Denotes zero or many characters
- _ Denotes one character
- Combination of % and_can also be used

Example 13:

(I) List the names of employees whose name starts with ‘s’

```
SELECT ENAME FROM EMP
WHERE ENAME LIKE ‘S%’;
```

(II) List the ename ending with ‘s’

```
SELECT ENAME FROM EMP
WHERE ENAME LIKE ‘%S’;
```

(III) List ename having I as a second character

```
SELECT ENAME FROM EMP
WHERE ENAME LIKE ‘_I%’;
```

d. **Is null operator**

Example 14:

to find employee whose manage-id is not specified

```
SELECT ENAME, MGR FROM EMP
WHERE MGR IS NULL;
```

8 **Logical Operators**

Rules of Precedence:

Order evaluated	Operator
1	All comparison operators

2	NOT
3	AND
4	OR

Example 15: To print those records of salesman or president who are having salary above 15000/-

```
Select ename, job, sal from emp
Where job = 'SALESMAN' or job = 'PRESIDENT'
And sal>15000;
```

The query formulation as above is incorrect for the problem statement. The correct formulation would be:

```
SELECT ENAME, JOB, SAL FROM EMP
WHERE (JOB = 'SALESMAN' OR JOB = 'PRESIDENT')
AND SAL>15000;
```

9. **Order by clause**

- It is used in the last portion of select statement
- By using this rows can be sorted
- By default it takes ascending order
- DESC: is used for sorting in descending order
- Sorting by column which is not in select list is possible
- Sorting by column Alias

Example 16:

```
SELECT EMPNO, ENAME, SAL*12 "ANNUAL"
FROM EMP
ORDER BY ANNUAL;
```

Example 17: Sorting by multiple columns; ascending order on department number and descending order of salary in each department.

```
SELECT ENAME, DEPTNO, SAL
FROM EMP
ORDER BY DEPTNO, SAL DESC;
```

10. **Aggregate functions**

- Some of these functions are count, min, max, avg.
- These functions help in getting consolidated information from a group of tuples.

Example 18: Find the total number of employees.

```
SELECT COUNT(*)
FROM EMP;
```

Example 19: Find the minimum, maximum and average salaries of employees of department D1.

```
SELECT MIN(SAL), MAX(SAL), AVG(SAL)
FROM EMP
WHERE DEPTNO = 'D1' ;
```

11. **Group By clauses**

- It is used to group database tuples on the basis of certain common attribute value such as employees of a department.
- WHERE clause still can be used, if needed.

Example 20: Find department number and Number of Employees working in that department.

```
SELECT DEPTNO, COUNT(EMPNO)
FROM EMP
GROUP BY DEPTNO;
```

Please note that while using group by and aggregate functions the only attributes that can be put in select clause are the aggregated functions and the attributes that have been used for grouping the information. For example, in the example 20, we cannot put ENAME attribute in the SELECT clause as it will not have a distinct value for the group. Please note the group here is created on DEPTNO.

12. Having clause

- This clause is used for creating conditions on grouped information.

Example 21: Find department number and maximum salary of those departments where maximum salary is more than Rs 20000/-.

```
SELECT DEPTNO, MAX(SAL)
FROM EMP
GROUP BY DEPTNO
HAVING MAX(SAL) > 20000;
```

INSERT INTO command:

- Values can be inserted for all columns or for the selected columns
- Values can be given through sub query explained in section 1.8
- In place of values parameter substitution can also be used with insert.
- If data is not available for all the columns, then the column list must be included following the table name.

Example 22: Insert the employee numbers, an increment amount of Rs.500/- and the increment date-today (which is being entered through function SYSDATE) of all the managers into a table INCR (increment due table) from the employee file.

```
INSERT INTO INCR
SELECT EMPNO, 500, SYSDATE FROM EMP
WHERE JOB = 'MANAGER';
```

Example 23: Insert values in a table using parameter substitution (& operator is used for it 1, 2 are the field numbers).

```
INSERT INTO EMP
VALUES (&1,'&2','&3', &4, &5, &6,NULL, &7);
```

Please note these values needs to be supplied at run time.

UPDATE Command:

Syntax is

```
UPDATE <table name>
SET <column name> = <value>
WHERE <condition>;
```

Sub query in the UPDATE command:

Example 24: Double the commission for employees, who have got at least 2 increments.

```
UPDATE EMP
SET COMM = COMM * 2
WHERE 2 <= (        SELECT COUNT (*) FROM INCR
WHERE INCR.EMPNO = EMP.EMPNO
GROUP BY EMPNO);
```

Please note the use of subquery that counts the number of increments given to each employee stored in the INCR table. Please note the comparison, instead of ……>=2, we have written reverse of it as 2 <= …..

DELETE Command

In the following example, the deletion will be performed in EMP table.No deletion will be performed in the INCR table.

Example 25: Delete the records of employees who have got no increment.

```
DELETE FROM EMP
WHERE EMPNO NOT IN (SELECT EMPNO FROM INCR);
```

☞ Check Your Progress 1

1) What are the advantages of SQL? Can you think of some disadvantages of SQL?

……

……

……

……..

2) Create the Room, Booking, and Guest tables using the integrity enhancement features of SQL with the following constraints:
   (a) Type must be one of Single, Double, or Family.
   (b) Price must be between Rs.100/- and Rs.1000/-.
   (c) roomNo must be between 1 and 100.
   (d) booking dateFrom and dateTo must be greater than today's date.
   (e) The same room cannot be double booked.
   (f) The same guest cannot have overlapping bookings.

……

……

……

……

……

……..

3) Define the function of each of the clauses in the SELECT statement. What are the restrictions imposed on these clauses?

……..

……..

……

……

……..

4) Consider the supplier relations.

S

SNO (Supplier Number)	SNAME (Supplier Name)	STATUS	CITY
S1	Prentice Hall	30	Calcutta
S2	McGraw Hill	30	Mumbai
S3	Wiley	20	Chennai
S4	Pearson	40	Delhi
S5	Galgotia	10	Delhi

SP

SNO	PNO (Part Number)	Quantity
S1	P1	300
S1	P2	200
S2	P1	100
S2	P2	400
S3	P2	200
S4	P2	200

a) Get supplier numbers for suppliers with status > 20 and city is Delhi
b) Get Supplier Numbers and status for suppliers in Delhi in descending order of status.
c) Get all pairs of supplier numbers such that the two suppliers are located in the same city. (Hint: It is retrieval involving join of a table with itself.)
d) Get unique supplier names for suppliers who supply part P2 without using IN operator.
e) Give the same query above by using the operator IN.
f) Get part numbers supplied by more than one supplier. (Hint : It is retrieval with sub-query block, with inter block reference and same table involved in both blocks)
g) Get supplier numbers for suppliers who are located in the same city as supplier S1. (Hint: Retrieval with sub query and unqualified comparison operator).
h) Get supplier names for suppliers who supply part P1. (Hint : Retrieval using EXISTS)
i) Get part numbers for parts whose quantity is greater than 200 or are currently supplied by S2. (Hint: It is a retrieval using union).
j) Suppose for the supplier S5 the value for status is NULL instead of 10. Get supplier numbers for suppliers greater than 25. (Hint: Retrieval using NULL).
k) Get unique supplier numbers supplying parts. (Hint: This query is using the built-in function count).
l) For each part supplied, get the part number and the total quantity supplied for that part. (Hint: The query using GROUP BY).
m) Get part numbers for all parts supplied by more than one supplier. (Hint: It is GROUP BY with HAVING).
n) For all those parts, for which the total quantity supplied is greater than 300, get the part number and the maximum quantity of the part supplied in a single supply. Order the result by descending part number within those maximum quantity values.
o) Double the status of all suppliers in Delhi. (Hint: UPDATE Operation).
p) Let us consider the table TEMP has one column, called PNO. Enter into TEMP part numbers for all parts supplied by S2.
q) Add part p7.
r) Delete all the suppliers in Mumbai and also the suppliers concerned.

1.5 DATA CONTROL

The data control basically refers to commands that allow system and data privileges to be passed to various users. These commands are normally available to database administrator. Let us look into some data control language commands:

Create a new user:

CREATE USER < user name > IDENTIFIED BY < Password>

Example 26:

CREATE USER MCA12 IDENTIFIED BY W123

Grant: It is used to provide database access permission to users. It is of two types (1) system level permission (2) Object level permission.

Example 27:

GRANT CREATE SESSION TO MCA12;
(This command provides system level permission on creating a session – not portable)

GRANT SELECT ON EMP TO MCA12;
(Object level permission on table EMP)

GRANT SELECT, INSERT, UPDATE, DELETE ON EMP TO MCA12;

GRANT SELECT, UPDATE ON EMP TO MCA12, MCA13;
(Two users)
GRANT ALL ON EMP TO PUBLIC;
(All permission to all users, do not use it. It is very dangerous for database)

Revoke: It is used to cancel the permission granted.

Example 28:

REVOKE ALL ON EMP FROM MCA12;
(All permissions will be cancelled)
You can also revoke only some of the permissions.

Drop: A user-id can be deleted by using drop command.

Example 29:

DROP USER MCA12;

Accessing information about permissions to all users

(1) Object level permissions: With the help of data dictionary you can view the permissions to user. Let us take the table name from oracle. In oracle the name of the table containing these permissions is user_tab_privs.
DESCRIBE USER_TAB_PRIVS ;
SELECT * FROM USER_TAB_PRIVS;

(2) System level permissions: With the help of data dictionary you can see them. Let us take the table name as user_sys_privs (used in oracle).
DESCRIBE USER_SYS_PRIVS ;

SELECT * FROM USER_SYS_PRIVS ;

All these commands are very specific to a data base system and may be different on different DBMS.

1.6 DATABASE OBJECTS: VIEWS, SEQUENCES, INDEXES AND SYNONYMS

Some of the important concepts in a database system are the views and indexes. Views are a mechanism that can support data independence and security. Indexes help in better query responses. Let us discuss about them along with two more concepts sequences and synonyms in more details.

1.6.1 Views

A view is like a window through which data from tables can be viewed or changed. The table on which a view is based is called Base table. The view is stored as a SELECT statement in the data dictionary. When the user wants to retrieve data, using view. Then the following steps are followed.

1) View definition is retrieved from data dictionary table. For example, the view definitions in oracle are to be retrieved from table name USER-VIEWS.
2) Checks access privileges for the view base table.
3) Converts the view query into an equivalent operation on the underlying base table

Advantages:

- It restricts data access.
- It makes complex queries look easy.
- It allows data independence.
- It presents different views of the same data.

Type of views:

Simple views and Complex Views

Feature	Simple Views	Complex Views
Number of tables	One	One or more
Contain Functions	No	Yes
Contain groups of data	No	Yes
Data Manipulation	IS allowed	Not always

Let us look into some of the examples. To see all the details about existing views in Oracle:

SELECT* FROM USER_VIEWS;

Creating a view:

- A query can be embedded within the CREATE VIEW STATEMENT
- A query can contain complex select statements including join, groups and sub-queries
- A query that defines the view **cannot contain** an order by clause.
- DML operation (delete/modify/add) cannot be applied if the view contains any of the following:

Delete (You can't delete if view contains following)	*Modify (you cannot modify if view contains following)*	*Insert (you cannot insert if view contains following)*
• Group functions • A group by clause • A distinct keyword	• Group functions • A group by clause • A distinct keyword • Columns defined by Expressions	• Group functions • A group by clause • A distinct keyword • Columns defined by Expressions • There are Not Null Columns in the base tables that are not selected by view.

Example 30: Create a view named employee salary having minimum, maximum and average salary for each department.

```
CREATE VIEW EMPSAL (NAME, MINSAL, MAXSAL, AVGSAL) AS
        SELECT D.DNAME, MIN(E.SAL),MAX(E.SAL),AVG(E.SAL)
        FROM EMP E, DEPT D
        WHERE E.DEPTNO=D.DEPTNO
         GROUP BY D.DNAME;
```

To see the result of the command above you can give the following command:

```
SELECT * FROM EMPSAL;
```

You may get some sample output like:

```
NAME           MINSAL    MAXSA     AVGSAL
--------------  ---------  ---------  -------------
ACCOUNTING      1300        5000       2916.6667
RESEARCH        800         3000       2175
SALES           950         2850       1566.6667
```

To see the structure of the view so created, you can give the following command:

```
DESCRIBE EMPSAL;
 Name                              Null?    Type
---------------                   ------------   ---------------------
 NAME                                        VARCHAR2 (14)
 MINSAL                                       NUMBER
 MAXSAL                                       NUMBER
 AVGSAL                                       NUMBER
```

Creating views with check option: This option restricts those updates of data values that cause records to go off the view. The following example explains this in more detail:

Example 31: To create a view for employees of Department = 30, such that user cannot change the department number from the view:

```
CREATE OR REPLACE VIEW EMPD30 AS
        SELECT EMPNO EMPL_NUM, ENAME NAME, SAL SALARY
        FROM EMP
        WHERE DEPTNO=30
        WITH CHECK OPTION;
```

Now the user cannot change the department number of any record of view EMPD30. If this change is allowed then the record in which the change has been made will go off the view as the view is only for department-30. This restricted because of use of WITH CHECK OPTION clause

Creating views with Read only option: In the view definition this option is used to ensure that no DML operations can be performed on the view.

Creating views with Replace option: This option is used to change the definition of the view without dropping and recreating it or regranting object privileges previously granted on it.

1.6.2 Sequences

Sequences:

- automatically generate unique numbers
- are sharable
- are typically used to create a primary key value
- replace application code
- speed up the efficiency of accessing sequence Values when cached in memory.

Example 32: Create a sequence named SEQSS that starts at 105, has a step of 1 and can take maximum value as 2000.

```
CREATE SEQUENCE SEQSS
START WITH 105
INCREMENT BY 1
 MAX VALUE 2000;
```

How the sequence so created is used? The following sequence of commands try to demonstrate the use of the sequence SEQSS.

Suppose a table person exists as:

```
SELECT * FROM PERSON;
Output:        CODE    NAME          ADDRESS
                ------  ------------- -------------
                  104     RAMESH    MUMBAI
```

Now, if we give the command:

```
INSERT INTO PERSON
        VALUES (SEQSS.NEXTVAL, &NAME, &ADDRESS)
```

On execution of statement above do the following input:

```
Enter value for name: 'Rakhi'
Enter value for address: 'New Delhi'
```

Now, give the following command to see the output:

```
SELECT * FROM PERSON;
  CODE NAME          ADDRESS
-------- -------------- ----------------
   104 RAMESH         MUMBAI
   105 Rakhi          NEW DELHI
```

The descriptions of sequences such as minimum value, maximum value, step or increment are stored in the data dictionary. For example, in oracle it is stored in the table user_sequences. You can see the description of sequences by giving the SELECT command.

Gaps in sequence values can occur when:

- A rollback occurs that is when a statement changes are not accepted.
- The system crashes
- A sequence is used in another table.

Modify a sequence:

```
ALTER SEQUENCE SEQSS
INCREMENT 2
MAXVALUE 3000;
```

Removing a sequence:

```
DROP SEQUENCE SEQSS;
```

1.6.3 Indexes and Synonyms

Some of the basic properties of indexes are:

- An Index is a schema Object
- Indexes can be created explicitly or automatically
- Indexes are used to speed up the retrieval of rows
- Indexes are logically and physically independent of the table. It means they can be created or dropped at any time and have no effect on the base tables or other indexes.
- However, when a table is dropped corresponding indexes are also dropped.

Creation of Indexes

Automatically: When a primary key or Unique constraint is defined in a table definition then a unique index is created automatically.

Manually: User can create non-unique indexes on columns to speed up access time to rows.

Example 33: The following commands create index on employee name and employee name + department number respectively.

```
CREATE INDEX EMP_ENAME_IDX ON EMP (ENAME);
CREATE INDEX EMP_MULTI_IDX ON EMP (ENAME, DEPTNO);
```

Finding details about created indexes: The data dictionary contains the name of index, table name and column names. For example, in Oracle a user-indexes and user-ind-columns view contains the details about user created indexes.

Remove an index from the data dictionary:

```
DROP INDEX EMP_ENAME_IDX;
```

Indexes cannot be modified.

Synonyms

It permits short names or alternative names for objects.

Example 34:

```
CREATE SYNONYM D30
FOR EMPD30;
```

Now if we give command:

```
SELECT * FROM D30;
```

The output will be:

NAME	MINSAL	MAXSAL	AVGSAL
ACCOUNTING	1300	5000	2916.6667
RESEARCH	800	3000	2175
SALES	950	2850	1566.6667

Removing a Synonym:

```
DROP SYNONYM D30;
```

1.7 TABLE HANDLING

In RDBMS more than one table can be handled at a time by using join operation. Join operation is a relational operation that causes two tables with a common domain to be combined into a single table or view. SQL specifies a join implicitly by referring the matching of common columns over which tables are joined in a WHERE clause. Two tables may be joined when each contains a column that shares a common domain with the other. The result of join operation is a single table. Selected columns from all the tables are included. Each row returned contains data from rows in the different input tables where values for the common columns match. An important rule of table handling is that there should be one condition within the WHERE clause for each pair of tables being joined. Thus if two tables are to be combined, one condition would be necessary, but if three tables (X, Y, Z) are to be combined then two conditions would be necessary because there are two pairs of tables (X-Y and Y-Z) OR (X-Z and Y-Z), and so forth. There are several possible types of joins in relational database queries. Four types of join operations are described below:

(1) **Equi Join**: A join in which the joining condition is based on equality between values in the common columns. Common columns appear (redundantly) in the result table. Consider the following relations:

- customer (custid, custname, ……..) and
- order (custid, ordered,……..)

Example 35: What are the names of all customers who have placed orders?

The required information is available in two tables, customer and order. The SQL solution requires joining the two table using equi join.

```
SELECT CUSTOMER.CUTOID, ORDER.CUSTOID,
                                    CUSTONAME, ORDERID
FROM CUSTOMER, ORDER
WHERE CUSTOMER.CUSTOID=ORDER.CUSTOID;
```

The output may be:

Customer.custoid	order.custoid	custoname	orderid
10	10	Pooja Enterprises	1001
12	12	Estern Enterprises	1002
3	3	Impressions	1003

(2) **Natural Join**: It is the same like Equi join except one of the duplicate columns is eliminated in the result table. The natural join is the most commonly used form of join operation.

Example 36:

```
SELECT CUSTOMER.CUTOID, CUSTONAME, ORDERID
FROM CUSTOMER, ORDER
WHERE CUSTOMER.CUSTOID=ORDER.CUSTOID;
```

Output:

custoid	custoname	orderid
10	Pooja Enterprises	1001
12	Estern Enterprises	1002
3	Impressions	1003

(3) **Outer Join**: The use of Outer Join is that it even joins those tuples that do not have matching values in common columns are also included in the result table. Outer join places null values in columns where there is not a match between

tables. A condition involving an outer join is that it cannot use the IN operator or cannot be linked to another condition by the OR operator.

Example 37: The following is an example of left outer join (which only considers the non-matching tuples of table on the left side of the join expression).

```
SELECT CUSTOMER.CUTOID, CUSTONAME, ORDERID
FROM CUSTOMER LEFT OUTER JOIN ORDER
WHERE CUSTOMER.CUSTOID =   ORDER.CUSTOID;
```

Output: The following result assumes a CUSTID in CUSTOMER table who have not issued any order so far.

CUSTOID	CUSTONAME	ORDERID
10	Pooja Enterprises	1001
12	Estern Enterprises	1002
3	Impressions	1003
15	South Enterprises	NULL

The other types of outer join are the Right outer join or complete outer join.

(4) **Self-Join**: It is a join operation where a table is joined with itself. Consider the following sample partial data of EMP table:

EMPNO	ENAME	MGRID	…..
1	Nirmal	4	
2	Kailash	4	
3	Veena	1	
4	Boss	NULL	
…..	…..	…	

Example 38: Find the name of each employee's manager name.

```
SELECT WORKER.ENAME || 'WORK FOR' || MANAGER.ENAME
FROM EMP WORKER, EMP MANAGER
WHERE WORKER.MGR=MANAGER.EMPNO;
```

Output:

Nirmal works for Boss
Kailash works for Boss
Veena works for Nirmal

☞ Check Your Progress 2

1) Discuss how the Access Control mechanism of SQL works.

……..

……..

……..

……..

2) Consider Hotel schema consisting of three tables Hotel, Booking and Guest,

```
CREATE TABLE Hotel
        hotelNo         HotelNumber         NOT NULL,
        hotelName       VARCHAR(20)         NOT NULL,
        city            VARCHAR(50)         NOT NULL,

        PRIMARY KEY (hotelNo));
```

```
CREATE TABLE Booking(
hotelNo         HotelNumbers        NOT NULL,
guestNo         GuestNumbers        NOT NULL,
dateFrom        BookingDate         NOT NULL,
dateTo          BookingDate         NULL,
roomNo          RoomNumber          NOT NULL,

PRIMARY KEY (hotelNo, guestNo, dateFrom),
FOREIGN KEY (hotelNo) REFERENCES Hotel
        ON DELETE CASCADE ON UPDATE CASCADE,
FOREIGN KEY (guestNo) REFERENCES Guest
        ON DELETE NO ACTION ON UPDATE CASCADE,
FOREIGN KEY (hotelNo, roomNo) REFERENCES Room
        ON DELETE NO ACTION ON UPDATE CASCADE);
CREATE TABLE Guest(
guestNo         GuestNumber         NOT NULL,
guestName       VARCHAR(20)         NOT NULL,
guestAddress    VARCHAR(50)         NOT NULL
PRIMARY KEY (guestno));

CREATE TABLE Room(
roomNo          RoomNumber          NOT NULL,
hotelNo         HotelNumbers        NOT NULL,
type            RoomType            NOT NULL DEFAULT 'S'
price           RoomPrice           NOT NULL,
PRIMARY KEY (roomNo, hotelNo),
FOREIGN KEY (hotelNo) REFERENCES Hotel
ON DELETE CASCADE ON UPDATE CASCADE);
```

Create a view containing the hotel name and the names of the guests staying at the hotel.

……

……

……

……

……

……

……

……

3) Give the users Manager and Director full access to views HotelData and BookingOutToday, with the privilege to pass the access on to other users.

……

……

……

……

……

1.8 NESTED QUERIES

By now we have discussed the basic commands including data definition and data manipulations. Now let us look into some more complex queries in this section.

Sub-queries

Some of the basic issues of sub-queries are:

- A sub-query is a SELECT statement that is embedded in a clause of another SELECT statement. They are often referred to as a NESTED SELECT or SUB SELECT or INNER SELECT.
- The sub-query (inner query) executes first before the main query. The result of the sub-query is used by the main query (outer query).
- Sub-query can be placed in WHERE or HAVING or FROM clauses.
- Format of using sub-queries:

  SELECT<select_list>
  FROM<table>
  WHERE expr OPERATOR
  (SELECT <select_list>
  FROM <TABLE>WHERE);
  Operator includes a comparison operator (single or multiple row operators)
  Single row operators: >, =, >=, <, <=, <>
  Multiple row operators: IN, ANY, ALL

- Order by clause cannot be used in sub-query, if specified it must be the last clause in the main select statement.
- Types of sub-queries:
  - Single row sub-query: It returns only one row from the inner select statement.
  - Multiple row sub-queries: it returns more than one row from the inner select statement
  - Multiple column sub-queries: it returns more than one column from the inner select statement.

  Single row operators are used with single row sub queries and multiple row operators are used with multiple row sub queries.
- The Outer and Inner queries can get data from different tables.
- Group Functions can be used in sub queries.

Consider the following partial relation EMP. Let us create some sub-queries for them

EMPNO	ENAME	JOB	SAL	DEPTNO
7566	Nirmal	MANAGER	2975	10
7788	Kailash	ANALYST	3000	10
7839	Karuna	PRESIDENT	5000	20
7902	Ashwin	ANALYST	3000	20
7905	Ashwini	MANAGER	4000	20

Example 39: Get the details of the person having the minimum salary.

```
SELECT ENAME, JOB, SAL
FROM EMP
 WHERE SAL =   (      SELECT MIN (SAL)
                      FROM EMP);
```

Output:

ENAME	JOB	SAL
Nirmal	MANAGER	2975

Example 40: Display the employees whose job title is the same as that of employee 7566 and salary is more than the salary of employee 7788.

```
SELECT ENAME, JOB
FROM EMP
WHERE JOB  =   (      SELECT JOB
                      FROM EMP
                      WHERE EMPPNO = 7566)
          AND SAL >  (  SELECT SAL
                      FROM EMP
                      WHERE EMPPNO=7788);
```

Output: Job title for the employee 7566 happens to be 'MANAGER')

ENAME	JOB
Ashwini	MANAGER

Having Clause with sub queries: First we recollect the GROUP BYclause. The following query finds the minimum salary in each department.

```
SELECT DEPTNO, MIN(SAL)
FROM.EMP
GROUP BY DEPTNO;
```

Output:

DEPTNO	SAL
10	2975
20	3000

Example 41: To find the minimum salary in those departments whose minimum salary is greater than minimum salary of department number 10.

```
SELECT DEPTNO, MIN(SAL)
FROM EMP
GROUP BY DEPTNO
HAVING MIN(SAL) >  (        SELECT MIN (SAL)
                            FROM EMP
                            WHERE DEPTNO = 10);
```

Output:

DEPTNO	SAL
20	3000

Example 42: Find the name, department number and salary of employees drawing minimum salary in that department.

```
SELECT ENAME, SAL, DEPTNO
FROM EMP
WHERE SAL IN (SELECT MIN (SAL)
                    FROM EMP
                    GROUP BY DEPTNO);
```

Output:

ENAME	SAL	DEPTNO
Nirmal	2975	10
Ashwin	3000	20

Find the salary of employees employed as an ANALYST
SELECT SAL FROM EMPWHERE JOB= ' ANALYST '

Output:

SAL
3000
3000

Example 43: Find the salary of employees who are not 'ANALYST' but get a salary less than or equal to any person employed as 'ANALYST'.

```
SELECT EMPNO, ENAME, JOB, SAL
FROMEMP
WHERE SAL <= ANY ( SELECT SAL
                   FROM EMP WHERE JOB = 'ANALYST' )
    AND JOB<>'ANALYST' ;
```

Output:

EMPNO	ENAME	JOB	SAL
7566	Nirmal	MANAGER	2975

Find the average salary in each department

SELECT DEPTNO, AVG(SAL) FROM EMP GROUP BY DEPTNO;
Result:

DEPTNO	SAL
10	2987.5
20	4000

Example 44: Find out the employee who draws a salary more than the average salary of all the departments.

```
SELECT EMPNO, ENAME, JOB, SAL
FROM EMP
WHERE SAL> ALL (SELECT AVG (SAL)
                FROM EMP
                GROUP BY DEPTNO);
```

Output:

EMPNO	ENAME	JOB	SAL
7839	Karuna	PRESIDENT	5000

Example 45: Find the employee name, salary, department number and average salary of his/her department, for those employees whose salary is more than the average salary of that department.

```
SELECT A.ENAME, A.SAL, A.DEPTNO, B.AVGSAL
FROM EMP A, (   SELECT DEPTNO, AVG (SAL)  AVGSAL
                FROM EMP
                GROUP BY DEPTNO) B
WHERE A.DEPTNO=B.DEPTNO AND A.SAL> B. AVGSAL;
```

Output:

ENAME	SAL	DEPTNO	AVGSAL
Kailash	3000	10	2987.5
Karuna	5000	20	4000

Multiple column Queries:
Syntax:

```
SELECT COLUMN1, COL2,……
FROM TABLE
WHERE (COLUMN1, COL2, …) IN
                (SELECT COLUMN1, COL2,….
                 FROM TABLE
                 WHERE <CONDITION>);
```

Example 46: Find the department number, name, job title and salary of those people who have the same job title and salary as those are in department 10.

```
SELECT DEPTNO,ENAME, JOB, SAL
FROM EMP
WHERE (JOB, SAL) IN  (      SELECT JOB, SAL
                            FROM EMP
                            WHERE DEPTNO=10);
```

Output:

DEPTNO	ENAME	JOB	SAL
10	Nirmal	MANAGER	2975
10	Kailash	ANALYST	3000
20	Ashwin	ANALYST	3000

☞ Check Your Progress 3

1) What is the difference between a sub-query and a join? Under what circumstances would you not be able to use a sub-query?

……

……

……

2) Use the Hotel schema defined in question number 2 (Check Your Progress 2) and answer the following queries:

- List the names and addresses of all guests in Delhi, alphabetically ordered by name.
- List the price and type of all rooms at the GRAND Hotel.
- List the rooms that are currently unoccupied at the Grosvenor Hotel.
- List the number of rooms in each hotel.
- What is the most commonly booked room type for hotel in Delhi?
- Update the price of all rooms by 5%.

……

……

……

3) Consider the following Relational database.

Employees (eno, ename, address, basic_salary)
Projects (Pno, Pname, enos-of-staff-alotted)
Workin (pno, eno, pjob)

Two queries regarding the data in the above database have been formulated in SQL. Describe the queries in English sentences.

(i)
```
SELECT ename
FROM employees
WHERE eno IN ( SELECT eno
                FROM workin
                GROUP BY eno
 HAVING COUNT (*) =  (SELECT COUNT (*) FROM projects));
```

(ii)
```
SELECT Pname
FROM projects
WHERE Pno IN ( SELECT Pno
FROM projects
MINUS
GROUP BY eno
(SELECT DISTINCT Pno FROM workin));
```

……..

……..

……

……..

……..

……

1.9 SUMMARY

This unit has introduced the SQL language for relational database definition, manipulation and control. The SQL environment includes an instance of an SQL DBMS with accessible databases and associated users and programmers. Each schema definition that describes the database objects is stored in data dictionary/ system catalog. Information contained in system catalog is maintained by the DBMS itself, rather than by the users of DBMS.

The data definition language commands are used to define a database, including its creation and the creation of its tables, indexes and views. Referential integrity constraints are also maintained through DDL commands. The DML commands of SQL are used to load, update and query the database. DCL commands are used to establish user access to the database. SQL commands directly affect the base tables, which contain the raw data, or they may affect database view, which has been created. The basic syntax of an SQL SELECT statement contains the following keywords: SELECT, FROM, WHERE, ORDER BY, GROUP BY and HAVING.

SELECT determines which attributes will be displayed in the query result table. FROM determines which tables or views will be used in the query. WHERE sets the criteria of the query, including any joins of multiple tables, which are necessary. ORDER BY determines the order in which the result will be displayed. GROUP BY is used to categorize results. HAVING is used to impose condition with GROUP BY.

1.10 SOLUTIONS / ANSWERS

Check Your Progress 1

1)

Advantages

- A standard for database query languages
- (Relatively) Easy to learn
- Portability
- SQL standard exists
- Both interactive and embedded access
- Can be used by specialist and non-specialist.

Yes, SQL has disadvantages. However, they are primarily more technical with reference to Language features and relational model theories. We are just putting them here for reference purposes.

Disadvantages

- Impedance mismatch – mixing programming paradigms with embedded access
- Lack of orthogonality – many different ways to express some queries
- Language is becoming enormous (SQL-92 is 6 times larger than predecessor)
- Handling of nulls in aggregate functions is not portable
- Result tables are not strictly relational – can contain duplicate tuples, imposes an ordering on both columns and rows.

2. CREATE DOMAIN RoomType AS CHAR(1) ……*[Constraint (a)]*
CHECK(VALUE IN (S, F, D));

CREATE DOMAIN HotelNumbers AS HotelNumber
CHECK(VALUE IN (SELECT hotelNo FROM Hotel));
[An additional constraint for hotel number for the application]

CREATE DOMAIN RoomPrice AS DECIMAL(5, 2)
CHECK(VALUE BETWEEN 1000 AND 10000);

CREATE DOMAIN RoomNumber AS VARCHAR(4)
CHECK(VALUE BETWEEN '1' AND '100');

[Constraint (c), one additional character is kept instead of 3 we have used 4characters but no space wastage as varchar]

```
CREATE TABLE Room(
        roomNo      RoomNumber      NOT NULL,
        hotelNo     HotelNumbers    NOT NULL,
        type        RoomType        NOT NULL DEFAULT S,
        price       RoomPrice       NOT NULL,
        PRIMARY KEY (roomNo, hotelNo),
        FOREIGN KEY (hotelNo) REFERENCES Hotel
                ON DELETE CASCADE ON UPDATE CASCADE);
        CREATE DOMAIN GuestNumber AS CHAR(4);
```

```
CREATE TABLE Guest(
        guestNo         GuestNumber         NOT NULL,
        guestName       VARCHAR(20)         NOT NULL,
        guestAddress    VARCHAR(50)         NOT NULL);

CREATE DOMAIN GuestNumbers AS GuestNumber
        CHECK(VALUE IN (SELECT guestNo FROM Guest));
                [A sort of referential constraint expressed within domain]

CREATE DOMAIN BookingDate AS DATETIME
        CHECK(VALUE > CURRENT_DATE);        [constraint (d) ]

CREATE TABLE Booking(
        hotelNo         HotelNumbers        NOT NULL,
        guestNo         GuestNumbers        NOT NULL,
        dateFrom        BookingDate         NOT NULL,
        dateTo          BookingDate         NULL,
        roomNo          RoomNumber          NOT NULL,
        PRIMARY KEY (hotelNo, guestNo, dateFrom),
        FOREIGN KEY (hotelNo) REFERENCES Hotel
                ON DELETE CASCADE ON UPDATE CASCADE,
        FOREIGN KEY (guestNo) REFERENCES Guest
                ON DELETE NO ACTION ON UPDATE CASCADE,
        FOREIGN KEY (hotelNo, roomNo) REFERENCES Room
                ON DELETE NO ACTION ON UPDATE CASCADE,
        CONSTRAINT RoomBooked
                CHECK (NOT EXISTS ( SELECT *
                                FROM Booking b
                                WHERE b.dateTo > Booking.dateFrom AND
                                b.dateFrom < Booking.dateTo AND
                                b.roomNo = Booking.roomNo AND
                                b.hotelNo = Booking.hotelNo)),
        CONSTRAINT GuestBooked
                CHECK (NOT EXISTS ( SELECT *
                                FROM Booking b
                                WHERE b.dateTo > Booking.dateFrom AND
                                b.dateFrom < Booking.dateTo AND
                                b.guestNo = Booking.guestNo)));
```

3.

FROM	Specifies the table or tables to be used.
WHERE	Filters the rows subject to some condition.
GROUP BY	Forms groups of rows with the same column value.
HAVING	Filters the groups subject to some condition.
SELECT	Specifies which columns are to appear in the output.
ORDER BY	Specifies the order of the output.

If the SELECT list includes an aggregate function and no GROUP BY clause is being used to group data together, then no item in the SELECT list can include any reference to a column unless that column is the argument to an aggregate function.

When GROUP BY is used, each item in the SELECT list must be single-valued per group. Further, the SELECT clause may only contain:

- Column names.
- Aggregate functions.
- Constants.
- An expression involving combinations of the above.

All column names in the SELECT list must appear in the GROUP BY clause unless the name is used only in an aggregate function.

3. Please note that some of the queries are sub-queries and queries requiring join. The meaning of these queries will be clearer as you proceed further with the Unit.

a)
```
SELECT SNO
FROM S
WHERE CITY = 'Delhi'
AND STATUS > 20;
```

Result:

SNO
S4

b)
```
SELECT SNO, STATUS
FROM S
WHERE CITY = 'Delhi'
ORDER BY STATUS DESC;
```

Result:

SNO	STATUS
S4	40
S5	10

c)
```
SELECT FIRST.SNO, SECOND.SNO
FROM S FIRST, S SECOND
WHERE FIRST.CITY = SECOND.CITY AND FIRST.SNO <
SECOND.SNO;
```

Please note that if you do not give the condition after AND you will get some unnecessary tuples such as: (S4, S4), (S5, S4) and (S5, S5).

Result:

SNO	SNO
S4	S5

d)
```
SELECT DISTINCT SNAME
FROM S, SP
WHERE S.SNO = SP.SNO
AND SP.PNO = 'P2';
```

Result:

SNAME
Prentice Hall

McGraw Hill
Wiley
Pearson

```
OR
SELECT SNAME
FROM S
WHERE SNO = ANY (   SELECT SNO
                    FROM SP
                    WHERE PNO = 'P2');
```

e)
```
SELECT SNAME
FROM S
WHERE SNO IN (      SELECT SNO
                    FROM SP
                    WHERE PNO = 'P2' );
```

f)
```
SELECT DISTINCT PNO
FROM SP SPX
WHERE PNO IN (      SELECT PNO
                    FROM SP
                WHERE SP.SNO = SPX.SNO AND SPX.SNO < SP.SNO );
```

This query can also be answered using count and group by. Please formulate that.

Result:

PNO
P1
P2

g)
```
SELECT SNO
FROM S
WHERE CITY = (      SELECT CITY
                    FROM S
                    WHERE SNO = 'S1' );
```

Result:

SNO
S1

h)
```
SELECT SNAME
FROM S
WHERE EXISTS (      SELECT *
                    FROM SP
                    WHERE SNO = S.SNO AND PNO = 'P1');
```

Result:

SNAME
Prentice Hall
McGraw Hill

i)
```
SELECT PNO
FROM SP
WHERE QUANTITY > 200 UNION (   SELECT PNO
                               FROM SP
                               WHERE SNO = S2 );
```

Result:

PNO
P1
P2

j)
```
SELECT SNO
FROM S
WHERE STATUS > 25 OR STATUS IS NULL;
```

Result:

SNO
S1
S2
S4
S5

k)
```
SELECT COUNT (DISTINCT SNO)
FROM SP;
```

Result: 4

l)
```
SELECT PNO, SUM(QUANTITY)
FROM SP
GROUP BY PNO;
```

Result:

PNO	SUM
P1	400
P2	1000

m)
```
SELECT PNO
FROM SP
GROUP BY PNO
HAVING COUNT(*) > 1 ;
```

The query is a same as that of part (f)

n)
```
SELECT PNO, MAX(QUANTITY)
FROM SP
WHERE QUANTITY > 200
GROUP BY PNO
HAVING SUM(QUANTITY) > 300
ORDER BY 2, PNO DESC;
```

o)
```
UPDATE  S
SET STATUS = 2 * STATUS
WHERE CITY = 'Delhi';
```

p)
```
INSERT INTO TEMP
SELECT PNO
FROM SP
WHERE SNO = 'S2';
```

q)
```
INSERT INTO SP( SNO,PNO,QUANTITY) < 'S5','P7',100> ;
```

Please note that part cannot be added without a supply in the present case.

Actually there should be another table for Parts

r)
```
DELETE S, SP
WHERE SNO = (     SELECT SNO
                  FROM S
                  WHERE CITY = 'Mumbai');
```

Check Your Progress 2

1) Each user has an authorization identifier (allocated by DBA).
Each object has an owner. Initially, only owner has access to an object but the owner can pass privileges to carry out certain actions on to other users via the GRANT statement and take away given privileges using REVOKE.

2)
```
CREATE VIEW HotelData(hotelName, guestName) AS
SELECT h.hotelName, g.guestName
        FROM Hotel  h, Guest g, Booking b
        WHERE h.hotelNo = b.hotelNo AND g.guestNo = b.guestNo AND
                b.dateFrom <= CURRENT_DATE AND
                b.dateTo >= CURRENT_DATE;
```

3)
```
GRANT ALL PRIVILEGES ON HotelData
        TO Manager, Director WITH GRANT OPTION;
GRANT ALL PRIVILEGES ON BookingOutToday
        TO Manager, Director WITH GRANT OPTION;
```

Check Your Progress 3

1) With a sub-query, the columns specified in the SELECT list are restricted to one table. Thus, cannot use a sub-query if the SELECT list contains columns from more than one table. But with a join operation SELECT list contains columns from more than two tables.

2) Answers of the queries are:

-
```
SELECT guestName, guestAddress FROM Guest
WHERE address LIKE '%Delhi%'
ORDER BY guestName;
```

-
```
SELECT price, type FROM Room
        WHERE hotelNo =
        (SELECT hotelNo FROM Hotel
        WHERE hotelName = 'GRAND Hotel');
```

-
```
SELECT * FROM Room r
WHERE roomNo NOT IN
(SELECT roomNo FROM Booking b, Hotel h
WHERE (dateFrom <= CURRENT_DATE AND
        dateTo >= CURRENT_DATE) AND
        b.hotelNo = h.hotelNo AND hotelName = 'GRAND Hotel');
```

-
```
SELECT hotelNo, COUNT(roomNo) AS count
FROM Room
GROUP BY hotelNo;
```

-
```
SELECT MAX(X)
```

```
FROM ( SELECT type, COUNT(type) AS X
FROM Booking b, Hotel h, Room r
WHERE r.roomNo = b.roomNo AND b.hotelNo = h.hotelNo AND
        city = 'LONDON'
GROUP BY type);
```

- UPDATE Room SET price = price*1.05;

3) (i) – Give names of employees who are working on all projects.

(ii) - Give names of the projects which are currently not being worked upon.

1.11 FURTHER READINGS

Fundamentals of DatabaseSystems; Almasri and Navathe; Pearson Education Limited; Fourth Edition; 2004.
A Practical Approach to Design, Implementation, and Management; Thomas Connolly and Carolyn Begg; Database Systems, Pearson Education Limited; Third Edition; 2004.
The Complete Reference; Kevin Lonely and George Koch; Oracle 9i, Tata McGraw-Hill; Fourth Edition; 2003.
Jeffrey A. Hoffer, Marry B. Prescott and Fred R. McFadden; Modern Database Management; Pearson Education Limited; Sixth Edition; 2004.

For Unit 9 : Please read the following unit of MCS-43 Block 1 Unit 3

UNIT 3 ADVANCED SQL

Structure **Page Nos.**

3.0 INTRODUCTION

The Structured Query Language (SQL) is a standard query language for database systems. It is considered as 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 described SQL in details in the course MCS-023 Block-2, Unit-1 where we discussed data definition, data manipulation, data control, queries, joins, group commands, 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 Procedure and Cursors. The concepts of embedded and dynamic SQL and SQLJ, which is used along with JAVA, are 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 in a SQL3 standard and will be applicable for any commercial database management system that supports SQL3 standards.

3.1 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.

3.2 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 within

the data or table creation statements with the help of statements 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 field or table. These are called assertions.

In this section we will discuss about 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 about them in more detail.

3.2.1 Assertions

Assertions are constraints that are normally of general nature. For example, the age of the student in a hypothetical University should not be more than 25 years or the minimum age of the teacher of that University should be 30 years. Such general constraints can be implemented with the help of an assertion statement. The syntax for creating assertion is:

Syntax:

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

Thus, the assertion on age for the University as above 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
                ));
```

The assertion name helps in identifying the constraints specified by the assertion. These names can be used to modify or delete an assertion later. But how are these assertions enforced? The database management system is responsible for enforcing the assertion on to the database such that the constraints stated in the assertion are not violated. Assertions are checked whenever a related relation changes.

Now try writing an assertion for a university system that stores a database of faculty as:

FACULTY (code, name, age, basic salary, medical-allow, other benefits)

MEDICAL-CLAIM (code, date, amount, comment)

Assertion: The total medical claims made by a faculty member in the current financial year should not exceed his/her medial allowance.

```
CREATE ASSERTION med-claim
CHECK (NOT EXISTS (
        SELECT code, SUM (amount), MIN(medical-allow)
        FROM (FACULTY NATRUAL JOIN MEDICAL-CLAIM)
        WHERE date>"31-03-2006"
        GROUP BY code
        HAVING MIN(medical-allow) < SUM(amount)
));
```

OR

```
CREATE ASSERTION med-claim
CHECK (NOT EXISTS (
        SELECT  *
        FROM FACULT f
        WHERE (f.code IN
        (SELECT code, SUM(amount)
        FROM MEDICAL-CLAIM m
WHERE date>"31-03-2006" AND f.code=m.code AND
f.medical-allow<SUM(amount)
```

Please analyse both the queries above and find the errors if any.

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.

3.2.2 Views

A view is a virtual table, which does not actually store data. But if it does not store any data, then what does it contain?

A view actually is a query and thus has a SELECT FROM WHERE ….. clause which works on physical table which stores the data. Thus, the view is a collection of relevant information for a specific entity. The 'view' has been introduced as a topic in MCS-023, Block 2, Unit-1. Let us recapitulate the SQL commands for creating views, with the help of an example.

Example: A student's database may have the following tables:

```
STUDENT  (name, enrolment-no, dateofbirth)
MARKS  (enrolment-no, 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 MCS-043.

```
CREATE VIEW SUBJECT-PERFORMANCE AS

        (SELECT  s.enrolment-no, name, subjectcode, smarks
        FROM STUDENT s, MARKS m
        WHERE s.enrolment-no = m.enrolment-no AND
        subjectcode 'MCS-043' ORDER BY s.enrolment-no;
```

A view can be dropped using a DROP statement as:

```
DROP VIEW SUBJECT-PERFORMANCE;
```

The table, which stores the data on which the statement of the view is written, is sometimes referred to as the base table. You can create views on two or more base tables by combining the data using joins. Thus, a view hides the logic of joining the tables from a user. You can also index the views too. This may speed up the performance. Indexed views may be beneficial for very large tables. 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 the views are 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. A query on this view may be: The teacher of the course MCS-043 wants to find the maximum and average marks in the course. The query for this in SQL will 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.enrolment-no=m.enrolment-no AND subjectcode= "MCS-043";
```

However, this approach has a major disadvantage. For a large database system, if complex queries have to be repeatedly executed on a view, the query modification will have to be done each time, 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, thus, materialising it. However, this strategy is not useful in situations where many database updates are made on the tables, which are used for view creation, as it will require suitable updating of a temporary table each time the base table is updated.

Can views be used for Data Manipulations?

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

A view allows data updating, if it follows the following conditions:

1) If the view is created from a single table, then:
   - For INSERT operation, the PRIMARY KEY column(s) and all the NOT NULL columns must be included in the view.
   - View should not be defined using any aggregate function or GROUP BY or HAVING or DISTINCT clauses. This is due to the fact that any update in such 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 table changing the class average marks for coursecode "MCS 043" 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 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. We can grant privileges to the 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 data entry user so that s/he may enter data on his/her behalf. The access rights can be revoked using the REVOKE statement as:

REVOKE ALL ON STUDENT-PERFORMANCE FROM ABC;

☞ **Check Your Progress 1**

1) Consider a constraint – the value of the age field of the student of 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 section 3.2.2.

……

……

……

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

……

……

……

3.3 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 type 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.

3.3.1 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 train booking for a train will not change. However, this query may be executed for many different trains. Please note that it will only change the input parameter to the query that is 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: Write a C program segment that prints the details of a student whose enrolment number is input.

Let us assume the relation

STUDENT (enrolno:char(9), name:Char(25), phone:integer(12), prog-code:char(3))

```
/* 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*/
/* 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 views statements.
- The distinction between an SQL statement and host language statement is made by using the key word EXEC SQL; thus, this key word helps in identifying the Embedded SQL statements by the pre-compiler.
- 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 of shared variables that are shared between the environments. Please note that enrolno[10], name[20], p-code[4]; etc. are shared variables, colon (:) declared in 'C'.

- 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 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 (:) this distinguishes whether we are referring to the shared host variable or an SQL attribute. Such similar names is 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 there typed 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'. The value of SQLCODE if less than 0 indicates an error. Similarly, SQLSTATE is a 5 char code the 6th char is for '\0' in the host language 'C'. Value "00000" in an SQLSTATE indicate no error. You can refer to 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 we can simply say

  DISCONNECT <name of the connection>;

However, these statements need to be checked in the commercial database management system, which you are using.

Execution of SQL query in the given program: To create the SQL query, first, the given value of enrolment number is transferred to SQL attribute value, the query then is executed and the result, which is a single tuple in this case, is transferred to shared host variables as indicated by the key word INTO after the SELECT statement.

The SQL query runs as a standard SQL query except the use of shared host variables. 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 a key to the relation, only one tuple will be transferred to the shared host variables. But what will happen if more than one tuple results on the execution of embedded SQL query. Such situations are handled with the help of a cursor. Let us discuss such queries in more detail.

3.3.2 Cursors and Embedded SQL

Let us first define the terminate 'cursor'. The Cursor can be defined as a pointer to the current tuple. It can also be defined as a portion of RAM allocated for the internal processing that has been set aside by the database server for database interactions. This portion may be used for query processing using SQL. But, what size of memory is to be allotted for the cursor? Ideally, the size allotted for the cursor should be equal to the memory required to hold the tuples that result from a query. However, the available memory puts a constraint on this size. Whenever a query results in a number of tuples, we 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 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 end of tuples.

In addition, cursors have some attributes through which we can determine the state of the cursor. These may be:

ISOPEN – It is true if 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: Write a C program segment that inputs the final grade of the students of MCA 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*/
/* program segment for the required function */
```

```
printf ("enter the programme code);
scanf ("%s, &p-code);
EXEC SQL DECLARE CURSOR GUPDATE
        SELECT enrolno, name, phone, grade
        FROM STUDENT
        WHERE progcode =: p-code
        FOR UPDATE OF grade;
EXEC SQL OPEN GUPDATE;
EXEC SQL FETCH FROM GUPDATE
        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 GUPDATE
        EXEC SQL FETCH FROM GUPDATE;
}
EXEC SQL CLOSE GUPDATE;
```

- 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 fetch into shared host variable followed by 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 the SQLCODE will be set by the last fetch statement executed just prior to while condition check.

How are these SQL statements compiled and error checked during embedded SQL?

- The SQL pre-compiler performs the type of checking of the various shared host variables to find any mismatches or errors on each of the SQL statements. It then stores the results into 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 we require dynamically embedded SQL also.

3.3.3 Dynamic SQL

Dynamic SQL, unlike embedded SQL statements, are 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 we 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: Write a dynamic SQL interface that allows a student to get and modify permissible details about him/her. The student may ask for 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 user names are enrolment number.

Solution: A sample program segment may be (please note that the syntax may change for different commercial DBMS).

```
/* declarations in SQL */
EXEC SQL BEGIN DECLARE SECTION;
        char inputfields (50);
        char tablename(10)
        char sqlquery ystring(200)
EXEC SQL END DECLARE SECTION;
        printf  ("Enter the fields you want to see \n");
        scanf ("SELECT%s", inputfields);
        printf ("Enter the name of table STUDENT or RESULT");
        scanf ("FROM%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 user name 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 or s/he can be suitably prompted.
- The query can be fabricated using a concatenation of strings. This is language dependent in the example and is not a portable feature in the present query.
- The query modification of the query is being done keeping security in mind.
- The query is prepared and executed using a suitable SQL EXEC commands.

3.3.4 SQLJ

Till now we have talked about embedding SQL in C, but how can we embed SQL statements into JAVA Program? For this purpose we use SQLJ. In SQLJ, a preprocessor called SQLJ translator translates SQLJ source file to JAVA source file. The JAVA file compiled and run on the database. 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.

Please note that SQLJ cannot use dynamic SQL. It can only use simple embedded SQL. SQLJ provides a standard form in which SQL statements can be embedded in

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:

Let us write a JAVA function to print the student details of student table, for the student who have taken admission in 2005 and name are like 'Shyam'. Assuming that the first two digits of the 9-digit enrolment number represents a year, the required input conditions may be:

- The enrolno should be more than "05000000" and
- The name contains the sub string "Shyam".

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 {
            if ( name equals ( " " ) )
                name =  "%";
            if (enrolno  equals (" ") )
                enrolno  = "%";
             SelRowIter       srows = null;
            # sql srows = { SELECT Enrolno, name, phone
                    FROM STUDENT
             WHERE   enrolno > :enrolno AND name like :name
              };
            while ( srows.next ( ) ) {
                int enrolno        =       srows. enrolno ( );
                String name        =       srows.name ( );

            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.to_string);
    }
}
```

☞ Check Your Progress 2

1) A University decided to enhance the marks for the students by 2 in the subject MCS-043 in the table: RESULT (enrolno, coursecode, marks). Write a segment of embedded SQL program to do this processing.

……

……

……

2) What is dynamic SQL?

……

……

……

3) Can you embed dynamic SQL in JAVA?

……

……

……

3.4 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 about them in more detail in this section.

3.4 .1 Stored Procedures

Stored procedures are collections of small programs that are stored in compiled form and have a specific purpose. For example, a company may have rules such as:

- A code (like enrolment number) with one of the digits as the check digit, which checks the validity of the code.
- Any date of change of value is to be recorded.

These rules are standard and need to be made applicable to all the applications that may be written. 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 it 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 in some commercial DBMS.

Syntax:

CREATE [or replace] PROCEDURE [user]PROCEDURE_NAME

[(argument datatype

[, argument datatype]….)]

BEGIN

Host Language statements;

END;

For example, consider a data entry screen that allows entry of enrolment number, first name and the last name of a student combines the first and last name and enters the complete name in upper case in the table STUDENT. The student table has the following structure:

STUDENT (enrolno:char(9), name:char(40));

The stored procedure for this may be written as:

```
CREATE PROCEDURE studententry (
        enrolment   IN  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    TRIM  =  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 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 error encountered by the most recently created procedure object.

You can also write an SQL command to display errors. The syntax of 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, etc. referred to by the procedure, 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.

3.4.2 Triggers

Triggers are somewhat similar to stored procedures except that they are activated automatically. When a trigger is 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 such that the desired action is executed.
- 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/constraints. An instance, of such a situation may be when an update in one relation affects only 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 Statement Level Triggers. Remember, that you must have proper access rights on the table on which you are creating the trigger. For example, you may have all the rights on a table or at least have UPDATE access rights on the tables, which are to be used in trigger. The following is the syntax of triggers in one of the commercial DBMS:

```
CREATE  TRIGGER <trigger_name>
[BEFORE | AFTER|
<Event>
ON <tablename>
[WHEN <condition> | FOR EACH ROW]
<Declarations of variables if needed is – 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:

Consider the following relation of a students 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 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 DBMS. Please refer to them in the respective DBMS for more details.

3.5 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 sources mentioned in 'further readings'.

SQL Interfaces: SQL also has a 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 the multiple databases.

SQL support for object-orientation: The latest SQL standard also supports the object-oriented features. It allows creation of abstract data types, nested relations, object identifies 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 stored procedure?

……

……

……

2) Write a trigger that restricts updating of STUDENT table outside the normal working hours/holiday.

……

……

……

3.6 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 states of the database. These constraints are expected to be satisfied by the database at all times. 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 a 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 tuple then we 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. The dynamic SQL is a way of creating queries through an application and compiling and executing them at the run time. Thus, it provides dynamic interface and hence the name.

Stored procedures are the procedures that are created for some application purpose. These procedures are precompiled procedures and can be invoked from application programs. Arguments can be passed to a stored procedure and it 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 books further readings list for more details relating to this unit.

3.7 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 we try to update the subjectcode. We cannot trace back a single tuple where such a change needs to take place. Please go through the rules for data updating though views.

Check Your Progress 2

1)

```
/*The table is RESULT (enrolno.coursecode, marks). */
EXEC SQL BEGIN DECLARE SECTION;
        char enrolno [10], coursecode[7]; /* grade is just one character*/
        int marks;
        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 grace 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;
}
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 ='MCS043';
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 resultmcs-43 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 first to create a valid query string and then to execute that query string. Since the queries are compiled and executed at the run time thus, it is slower than simple embedded SQL.

3) No, at present JAVA cannot use dynamic SQL.

Check Your Progress 3

1) Stored procedure is a compiled procedure in a host language that has been written for some 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, syntax of which will be different in different RDBMS.