UNIT 1 RELATIONAL DATABASE DESIGN

Structure **Page Nos.**

1.0	Introduction	7
1.1	Objectives	7
1.2	Features of Good Database Design	7
1.3	Enhanced ER Tools	8
1.4	Functional Dependency: Theory and Normalisation	13
1.5	Multivalued Dependency and Fourth Normal Form	15
1.6	Join Dependencies and Fifth Normal Form/PJNF	19
1.7	Inclusion Dependencies and Template Dependencies	23
1.8	Domain Key Normal Form (DKNF)	25
1.9	Modeling Temporal Data	26
1.10	Summary	27
1.11	Solutions/Answers	27
1.12	Further Readings	29

1.0 INTRODUCTION

This unit provides a detailed discussion of some of the tools and theories of good database design. The ER modeling concepts discussed in MCS-023 Block 1 Unit 2 are sufficient for representing many database schemas for database applications. However, the advanced applications of the database technologies are more complex. This resulted in extension of the ER model to EER model. However, to define the concept of EER model you must go through the concepts of E-R model first (in the above mentioned block) as these concepts are interrelated. This unit also discusses the concepts of functional, multi-valued and join dependencies and related normal forms. Some advance dependencies and a brief introduction about temporal data have also been covered in the unit.

1.1 OBJECTIVES

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

- define a good database design;
- draw a simple EER diagram;
- describe functional dependency and normalisation;
- explain the multi-valued dependency and 4NF;
- define join dependency and 5NF;
- define inclusion dependencies and Template Dependency, and
- define temporal data.

1.2 FEATURES OF GOOD DATABASE DESIGN

A good database design has the following features:

- ***Faithfulness:*** The design and implementation should be faithful to the requirements.
    - The use of *constraints* helps to achieve this feature.
- ***Avoid Redundancy:*** Something is redundant if when hidden from view, you could still figure it out from other data. This value is important because redundancy.

- wastes space and
- leads to inconsistency.

- ***Simplicity:*** Simplicity requires that the design and implementation avoid introducing more elements than are absolutely necessary – Keep it Simple (KIS).
  - This value requires designers to avoid introducing unnecessary intermediate concepts.
- ***Right kind of element:*** Attributes are easier to implement but entity sets and relationships are necessary to ensure that the right kind of element is introduced.

1.3 ENHANCED ER TOOLS

In addition to ER modeling concepts the EER model includes:

1) Subclass and Super class
2) Inheritance
3) Specialisation and Generalisation.

To describe the concepts of **subclass *and* super class** first let us revisit the concept of 'entity'. The basic object that an E-R model represents is an entity, which is a "thing" in the real world with an independent existence. An entity may be an object with a physical existence or it may be an object with a conceptual existence. Each entity has attributes (the particular properties that describe it).

For Example, the entity *vehicle* describes the type (that is, the attributes and relationship) of each *vehicle* entity and also refers to the current set of *vehicle* entities in the showroom database. Some times to signify the database application various meaningful sub-groupings of entity is done explicitly. For example, the members of the entity *vehicle* are further meaningfully sub- grouped as: Car, Scooter, truck and so on.

The set of entities in each of the groupings is a subset of the entities that belongs to the entity set *vehicle*. In other words every sub-grouping must be *vehicle*. Therefore, these sub-groupings are called a subclass of the *vehicle* entity type and the *vehicle* itself is called the super class for each of these subclasses.

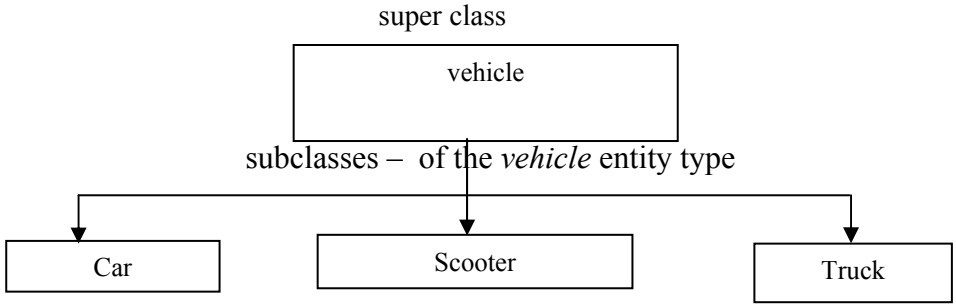

Figure 1: A class hierarchy

The relationship between a super class and any of its subclasses is called class/subclass relationship. It is often called an IS-A or relationship because of the way we refer to the concept, we say, "car ***is-a*** vehicle". The member entity of the subclass represents the same real world as the member entity of the super class. If an entity is a member of a subclass, by default it must also become a member of the super class whereas it is not necessary that every entity of the super class must be a member of its subclass. From the discussion above on sub/super classes we can say that an entity that is a member of a subclass inherits all the attributes of

the entity as a member of the super class. Notice that the type of an entity is defined by the attributes it possesses and the relationship types in which it participates; therefore, the entity also inherits all the relationships in which the super class participates. According to **inheritance** the subclass has its own attributes and relationships together with all attributes and relationships it inherits from the super class.

The process of defining the subclasses of an entity type is called **specialisation**, where the entity type is called the super class of the specialisation. The above said specialised set of subclasses are defined on the basis of some common but distinguishing characteristics of the entities in the super class. For example, the set of subclasses (car, scooter, truck) is a specialisation of the super class *vehicle* that distinguished among vehicles entities based on the vehicle type of each entity. We may have several other specialisations of the same entity type based on different common but distinctive characteristics. *Figure 2* shows how we can represent a specialisation with the help of an EER diagram.

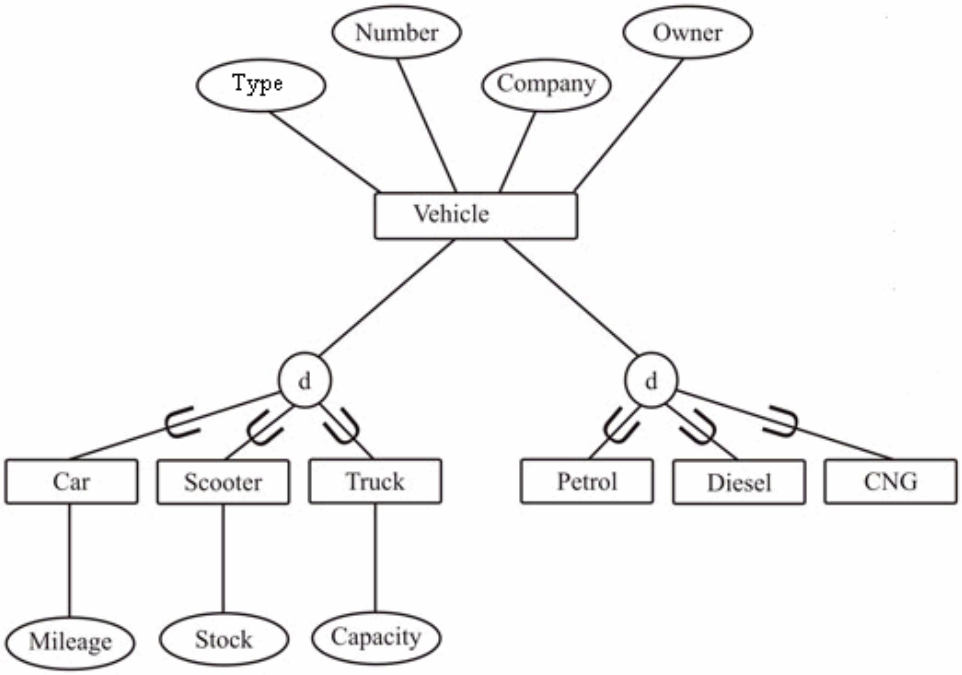

Figure 2: EER diagram representing specialisation

The subclasses that define a specialisation are attached by lines to a circle, which is connected further with the super class. The circle connecting the super class with the subclass indicates the direction of the super class/ subclass relationship. The letter 'd' in the circle indicates that all these subclasses are **disjoint** constraints.

Attributes that apply only to entities of a particular subclass – such as mileage of car, stock of scooter and capacity of truck are attached to the rectangle representing that subclass. Notice that an entity that belongs to a subclass represents the same real-world entity as the entity connected to super class, even though the same entity is shown twice – one in the subclass and the other in the super class. A subclass is defined in order to group the entities to which these attributes apply. The members of a subclass may still share the majority of their attributes with the other members of the super class (as shown in *Figure 3*).

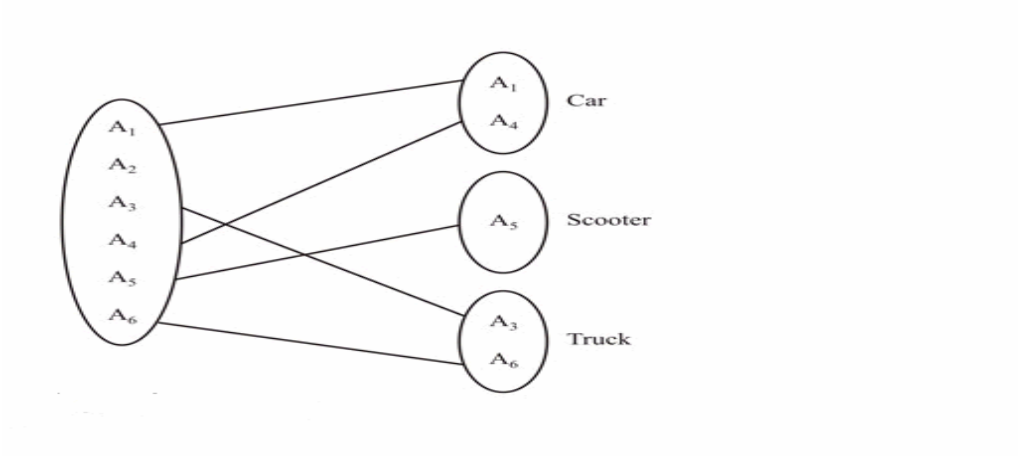

Figure 3: Sharing of members of the super class vehicle and its subclasses

Hence the specialisation is a set of subclasses of an entity type, which establishes additional specific attributes with each subclass and also establishes additional specific relationship types between each subclass and other entity types or other subclasses.

Generalisation is the reverse process of specialisation; in other words, it is a process of suppressing the differences between several entity types, identifying their common features into a single super class. For example, the entity type CAR and TRUCK can be generalised into entity type VEHICLE. Therefore, CAR and TRUCK can now be subclasses of the super class generalised class VEHICLE.

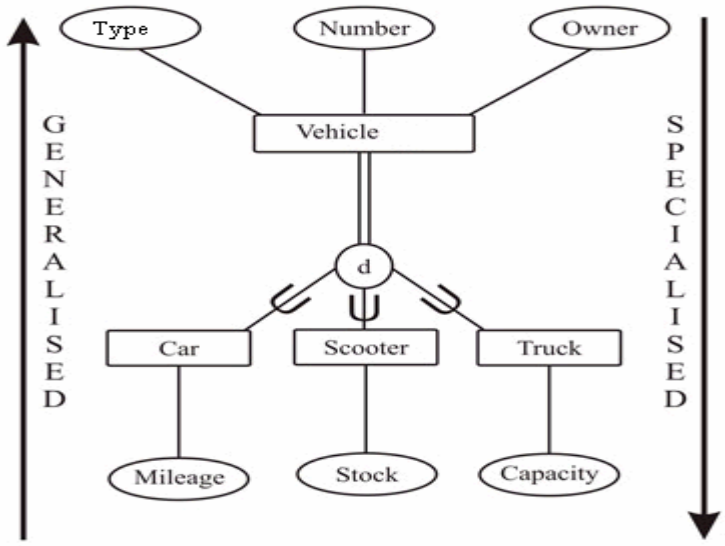

Figure 4: Generalisation and specialisation

Constraints and Characteristics of Specialisation and Generalisation

A super class may either have a single subclass or many subclasses in specialisation. In case of only one subclass we do not use circle notation to show the relationship of subclass/super class. Sometimes in specialisation, the subclass becomes the member of the super class after satisfying a condition on the value of some attributes of the super class. Such subclasses are called *condition-defined* subclasses or predicate defined subclasses. For example, vehicle entity type has an attribute vehicle "type", as shown in the *Figure 4.*

We can specify the condition of membership for a subclass – car, truck, scooter – by the predicate – vehicle 'type' of the super class vehicle. Therefore, a vehicle object

can be a member of the sub-class, if it satisfies the membership condition for that sub-class. For example, to be a member of sub-class car a vehicle must have the condition vehicle "type = car" as true. A specialisation where all the sub-classes have the membership condition defined on the same attribute of the super class, is called an *attribute-defined* specialisation. The common attribute that defines the condition is called the *defining attribute* of the specialisation. If no condition is specified to determine the membership of subclass in specialisation then it is called user-defined, as in such a case a database user must determine the membership.

Disjointness is also the constraints to a specialisation. It means that an entity can be a member of at most one of the subclasses of the specialisation. In an attribute-defined specialisation the disjointness constraint means that an entity can be a member of a single sub-class only. In the *Figure 4,* the symbol 'd' in circle stands for disjoint.

But if the real world entity is not disjoint their set of entities may overlap; that is an entity may be a member of more than one subclass of the specialisation. This is represented by an (o) in the circle. For example, if we classify cars as luxury cars and cars then they will overlap.

When every entity in the super class must be a member of some subclass in the specialisation it is called total specialisation. But if an entity does not belong to any of the subclasses it is called partial specialisation. The total is represented by a double line.

This is to note that in specialisation and generalisation the deletion of an entity from a super class implies automatic deletion from subclasses belonging to the same; similarly insertion of an entity in a super class implies that the entity is mandatorily inserted in all attribute defined subclass for which the entity satisfies the defining predicate. But in case of total specialisation, insertion of an entity in a super class implies compulsory insertion in at least one of the subclasses of the specialisation.

In some cases, a single class has a similar relationship with more than one class. For example, the sub class 'car' may be owned by two different types of owners: INDIVIDUAL or ORGANISATION. Both these types of owners are different classes thus such a situation can be modeled with the help of a Union.

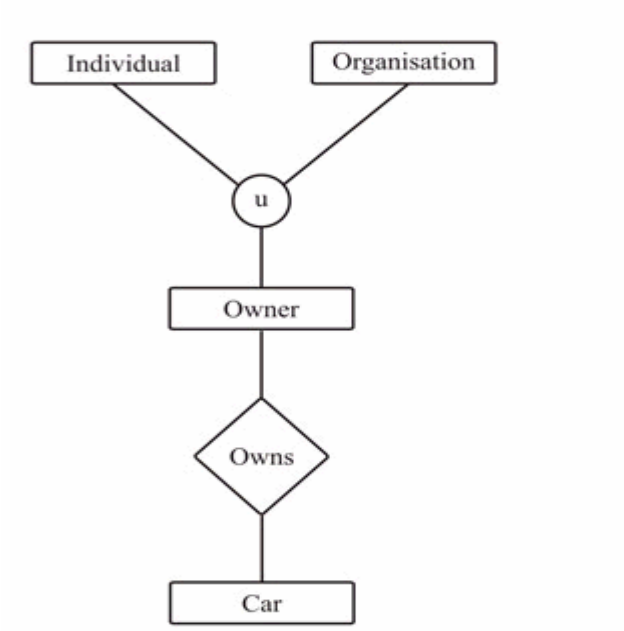

Figure 5: Union of classes

Converting EER diagram to table

The rules for converting EER diagram – which primarily includes specialisation and generalisation hierarchy are the same as in ER diagram. Let us recapitulate the rules:

- Create a table for each strong entity set.
- Create a table for a weak entity set; however, include the primary key of the strong entity on which it depends in the table.
- Create a table for each binary m : n relationship set having the primary keys of both the participating entities; however, for a binary m : 1 or 1 : m relationship, the primary key on the m side should be used as the primary key. For binary 1:1 relationship set the primary key is chosen from any of the participating relations.
- Composite attribute may sometimes be converted to a separate table.
- For generalisation or specialisation hierarchy a table can be created for higher level and each of the lower level entities. The higher level entity would have the common attributes and each lower level table would have the primary key of higher level entity and the attributes defined at the lower specialised level. However, for a complete disjoint hierarchy no table is made at the higher level, but the tables are made at the lower level including the attributes of higher level.
- For an aggregation, all the entities and relationships of the aggregation are transformed into the table on the basis of the above rules. The relationship that exists between the simple entity and the aggregated entity, have the primary key of the simple entity and the primary key of the relationship of the aggregated entity.

So let us now discuss the process of converting EER diagram into a table. In case of disjoint constrains with total participation. It is advisable to create separate tables for the subclasses. But the only problem in such a case will be to see that referential entity constraints are met suitably.

For example, EER diagram at *Figure 4* can be converted into a table as:

CAR (**Number**, owner, type, mileage)
SCOOTER (**Number**, owner, type, stock)
TRUCK (**Number,** owner, type, capacity)

Please note that referential integrity constraint in this case would require relationship with three tables and thus is more complex.

In case there is no total participation in the *Figure* 4 then there will be some vehicles, which are not car, scooter and truck, so how can we represent these? Also when overlapping constraint is used, then some tuples may get represented in more than one table. Thus, in such cases, it is ideal to create one table for the super class and the primary key and any other attribute of the subclass. For example, assuming total participation does not exist in *Figure* 4, then, a good table design for such a system may be:

VEHICLE (**Number,** owner, type)
CAR (**Number,** mileage)
SCOOTER (**Number,** stock)
TRUCK (**Number,** capacity)

Finally, in the case of union since it represents dissimilar classes we may represent separate tables. For example, both individual and organisation will be modeled to separate tables.

☞ **Check Your Progress 1**

1) What is the use of EER diagram?

……

……

……

2) What are the constraints used in EER diagrams?

……

……

……

3) How is an EER diagram converted into a table?

……

……

……

1.4 FUNCTIONAL DEPENDENCY: THEORY AND NORMALISATION

When a single constraint is established between two sets of attributes from the database it is called *functional dependency*. We have briefly discussed this in MCS-023. Let us discuss it in some more detail, especially with respect to formal theory of data dependencies. Let us consider a single universal relation scheme "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 relation state of "A". The constraint is that, for any two tuples t1 and t2 in "A" that have t1(X) = t2 (X), we must also have t1(Y) = t2(Y). It means that, if tuple t1 and t2 have same values for attributes X then X→Y to hold t1 and t2 must have same values for attributes Y.

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 two tuples in "A", agree on their X value then
2) they **must** agree on their Y value.

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

This 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 codes in India, with the proliferation of mobile telephone, the FD is no longer true.

Consider a relation

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

We know that the following functional dependencies (we identify these primarily from constraints, there is no thumb rule to do so otherwise) should hold:

- **enrolno →sname** (the enrolment number of a student uniquely determines the student names alternatively, we 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).

These FDs 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 reading list. You have already studied the functional dependence (FDs) and its use in normalisation till BCNF in MCS-023. However, we will briefly define the normal forms.

Normalisation

The first concept of normalisation was proposed by Mr. Codd in 1972. Initially, he alone proposed three normal forms named first, second and third normal form. Later on, with the joint efforts of Boyce and Codd, a stronger definition of 3NF called Boyce-Codd Normal Form (BCNF) was proposed. All the said normal forms are based on the functional dependencies among the attributes of a relation. 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 is based on the two assumptions /information above. Codd takes a relation schema through a series of tests to ascertain whether it satisfies a certain normal form. The 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. You have already gone through this process in MCS-023.

Later on, fourth normal form (4NF) and a fifth normal form (5NF) were proposed based on the concept of multivalued dependencies and join dependencies respectively. We will discuss these in more detail in the subsequent sections.

Therefore, normalisation 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 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.

The degree of normal forms to which a relation schema has been normalised through decomposition confirm the existence of additional properties that the relational schemas should possess. It could include any or both of two properties.

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

Based on performance reasons, relations may be left in a lower normalisation status. It is not mandatory that the database designer must normalise to the highest possible normal form. The process of storing the join of higher normal form relations, as a base relation (which is in a lower normal form) is known as denormalisation).

1.5 MULTIVALUED DEPENDENCIES AND FOURTH NORMAL FORM

In database modeling using the E-R Modeling technique, we usually face known difficulties that may arise when an entity has multivalued attributes. In the relational model, if all of the information about such entity is to be represented in one relation, it will be necessary to repeat all the information other than the multivalue attribute value to represent all the information. It will result in multi-tuples about the same instance of the entity in the relation and the relation having a composite key (the entity id and the mutlivalued attribute). This situation becomes much worse if an entity has more than one multivalued attributes and these values are represented in one relation by a number of tuples for each entity instance such that every value of one of the multivalued attributes appears with every value of the second multivalued attribute to maintain consistency. The multivalued dependency relates to this problem when more than one multivalued attributes exist. Let us consider the same through an example relation that represents an entity 'employee'.

emp (e#, dept, salary, job)

We have so far considered normalisation based on functional dependencies that apply only to single-valued facts. For example, $e\# \rightarrow dept$ implies only one *dept* value for each value of *e#*. Not all information in a database is single-valued, for example, *job* in an employee relation may be the list of all projects that the employee is currently working on. Although *e#* determines the list of all the projects that an employee is working on, yet, $e\# \rightarrow job$ is not a functional dependency.

The fourth and fifth normal forms deal with multivalued dependencies. Before discussing the 4NF and 5NF we will discuss the following example to illustrate the concept of multivalued dependency.

programmer (emp_name, projects, languages)

The above relation includes two multivalued attributes of the entity *programmer* - *projects* and *languages*. There are no functional dependencies.

The attributes *projects* and *languages* are assumed to be independent of each other. If we were to consider *projects* and *languages* as separate entities, we would have two relationships (one between *employees* and *projects* and the other between *employees* and programming *languages*). Both the above relationships are many-to-many relation, in the following sense:

1) one programmer could have several projects, and

2) may know several programming languages, also

3) one project may be obtained by several programmers, and

4) one programming language may be known to many programmers.

The above relation is in 3NF (even in BCNF) with some disadvantages. Suppose a programmer has several projects (Proj_A, Proj_B, Proj_C, etc.) and is proficient in several programming languages, how should this information be represented? There are several possibilities.

Emp_name	Projects	languages
DEV	Proj_A	C
DEV	Proj_A	JAVA
DEV	Proj_A	C++
DEV	Proj_B	C
DEV	Proj_B	JAVA
DEV	Proj_B	C++

emp_name	Projects	languages
DEV	Proj_A	NULL
DEV	Proj_B	NULL
DEV	NULL	C
DEV	NULL	JAVA
DEV	NULL	C++

emp_name	Projects	languages
DEV	Proj_A	C
DEV	Proj_B	JAVA
DEV	NULL	C++

Other variations are possible. Please note this is so as there is no relationship between the attributes ‘projects’ and programming ‘languages’. All the said variations have some disadvantages. If the information is repeated, we face the problems of repeated information and anomalies as we did when second or third normal form conditions were violated. Without repetition, difficulties still exist with insertions, deletions and update operations. For example, in the first table we want to insert a new person RAM who has just joined the organisation and is proficient in C and JAVA. However, this information cannot be inserted in the first table as RAM has not been allotted to work on any project. Thus, there is an insertion anomaly in the first table. Similarly, if both Project A and Project B get completed on which DEV was working (so we delete all the tuples in the first table) then the information that DEV is proficient in C, JAVA, and C++ languages will also be lost. This is the deletion anomaly. Finally, please note that the information that DEV is working on Project A is being repeated at least three times. Also the information that DEV is proficient in JAVA is repeated. Thus, there is redundancy of information in the first tables that may lead to inconsistency on updating (update anomaly).

In the second and third tables above, the role of NULL values is confusing. Also the candidate key in the above relations is (emp name, projects, language) and existential integrity requires that no NULLs be specified. These problems may be overcome by decomposing a relation as follows:

emp_name	Projects
DEV	Proj_A
DEV	Proj_B

emp_name	languages
DEV	C
DEV	JAVA
DEV	C++

This decomposition is the concept of 4NF. Functional dependency $A \rightarrow B$ relates one value of A to one value of B while multivalued dependency $A \rightarrow\rightarrow B$ defines a relationship where a set of values of attribute B are determined by a single value of A. Multivalued dependencies were developed to provide a basis for decomposition of relations like the one above. Let us define the multivalued dependency formally.

Multivalued dependency: *The multivalued dependency $X \rightarrow\rightarrow Y$ is said to hold for a relation R(X, Y, Z) if, for a given set of value (set of values if X is more than one attribute) for attribute X, there is a set of (zero or more) associated values for the set of attributes Y and the Y values depend only on X values and have no dependence on the set of attributes Z.*

Please note that whenever $X \rightarrow\rightarrow Y$ holds, so does $X \rightarrow\rightarrow Z$ since the role of the attributes Y and Z is symmetrical.

In the example given above, if there was some dependence between the attributes *projects* and *language*, for example, the language was related to the projects (perhaps the projects are prepared in a particular language), then the relation would not have MVD and could not be decomposed into two relations as above. However, assuming there is no dependence, *emp_name* $\rightarrow\rightarrow$ *projects and emp_name* $\rightarrow\rightarrow$ *languages* holds.

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. If a relation like *emp(eno, edpendent#)* has a relationship between *eno* and *edependent#* in which *eno* uniquely determines the **values** of *edependent#,* the dependence of *edependent#* on *eno* is called a trivial MVD since the relation *emp* cannot be decomposed any further.

Therefore, a relation having non-trivial MVDs must have at least three attributes; two of them 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 in the relation.

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:

t1X] = t2[X] = t3[X] = t4[X]
t3[Y] = t1[Y] and *t3[Z] = t2[Z]*
t4[Y] = t2[Y] and *t4[Z] = t1[Z]*

In other words if *t1* and *t2* are given by:

t1 = [X, Y1, Z1], and

t2 = [X, Y2, Z2]

then there must be tuples *t3* and *t4* such that:

t3 = [X, Y1, Z2], and

t4 = [X, Y2, Z1]

We are, therefore, insisting 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 multivalue dependent on *X* then *Z1 <> Z2*).

The theory of multivalued dependencies is very similar to that for functional dependencies. Given *D* a set of MVDs, we may 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.

We have considered an example of *Programmer(Emp name, projects, languages)* and discussed the problems that may arise if the relation is not normalised further. We also saw how the relation could be decomposed into *P1(emp name, projects)* and *P2(emp name, languages)* to overcome these problems. The decomposed relations are in fourth normal form (4NF), which we shall now define.

We now define 4NF. A relation *R* is in 4NF if, whenever a multivalued dependency $X \rightarrow\rightarrow Y$ holds, then either

(a) the dependency is trivial, or

(b) *X* is a candidate key for *R*.

The dependency $X \rightarrow\rightarrow \phi$ or $X \rightarrow\rightarrow Y$ in a relation *R (X, Y)* is trivial, since they must hold for all *R (X, Y)*. Similarly, in a trivial MVD *(X, Y)* $\rightarrow\rightarrow Z$ must hold for all relations *R (X, Y, Z)* with only three attributes.

If a relation has more than one multivalued attribute, we 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 all dependencies are a result of keys. 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.

1.6 JOIN DEPENDENCIES AND 5NF /PJNF

Based on the discussion above, we know that the normal forms require that the 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 rare cases, a relation can have problems like redundant information and update 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 using the 5NF. But 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.

The fifth normal form 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 satisfies join dependency *(R_1, R_2, ..., R_n) if and only if *R* is equal to the join of R_1, R_2, ..., R_n where R_i are subsets of the set of attributes of *R*.

A relation *R* is in 5NF if for all join dependencies at least one of the following holds:

(a) (R_1, R_2, ..., R_n) is a trivial join-dependency (that is, one of R_i is R)

(b) Every R_i is a candidate key for R.

An example of 5NF can be provided by the same above example that deals with emp_name, Projects and Programming languages with some modifications:

emp_name	Projects	Languages
DEV	Proj_A	C
RAM	Proj_A	JAVA
DEV	Proj_B	C
RAM	Proj_B	C++

The relation above assumes that any employee can work on any project and knows any of the three languages. The relation also says that any employee can work on projects Proj_A, Proj_B, Proj_C and may be using a different programming languages in their projects. No employee takes all the projects and no project uses all the programming languages and therefore all three fields are needed to represent the information. Thus, all the three attributes are independent of each other.

The relation above does not have any FDs and MVDs since the attributes emp_name, project and languages are independent; they are related to each other only by the pairings that have significant information in them. For example, DEV is working on Project A using C languague. Thus, the key to the relation is (emp_name, project, languague). The relation is in 4NF, but still suffers from the insertion, deletion, and update anomalies as discussed for the previous form of this relation. However, the relation therefore cannot be decomposed in two relations.

(emp_name, project), and

(emp_name, language)

Why?

Let us explain this with the help of a definition of join dependency. The decomposition mentioned above will create tables as given below:

Emp_project

emp_name	Projects
DEV	Proj_A
RAM	Proj_A
DEV	Proj_B
RAM	Proj_B

Emp_language

Emp_name	languages
DEV	C
RAM	JAVA
RAM	C++

On taking join of these relations on emp_name it will produce the following result:

emp_name	Projects	Languages
DEV	Proj_A	C
RAM	Proj_A	JAVA
RAM	Proj_A	C++
DEV	Proj_B	C
RAM	Proj_B	JAVA
RAM	Proj_B	C++

Since the joined table does not match the actual table, we can say that it is a lossy decomposition. Thus, the expected join dependency expression:

*((emp_name, project), (emp_name, language)) does not satisfy the conditions of lossless decomposition. Hence, the decomposed tables are losing some important information.

Can the relation 'Programmer' be decomposed in the following three relations?

(emp_name, project),
(emp_name, language) and
(Projects, language)

Please verify whether this decomposition in lossless or not. The join dependency in this case would be:

*((emp_name, project), (emp_name, language), (project, language))
and it can be shown that this decomposition is lossless.

Project-Join Normal Form

(Reference website: http://codex.cs.yale.edu/avi/db-book/online-dir/c.pdf)
PJNF is defined using the concept of the join dependencies. A relation schema *R* having a set *F* of functional, multivalued, and join dependencies, is in PJNF (5 NF), if for all the join dependencies in the closure of F (referred to as *F*+) that are of the form

*(*R1, R2, . . .,Rn*), where each *Ri* ⊆ *R* and *R* = *R*1 ∪ *R*2 ∪ . . . ∪ *Rn*, at least one of the following holds:

- *(*R1, R2, . . .,Rn*) is a trivial join dependency.
- Every *Ri* is a superkey for *R*.

PJNF is also referred to as the Fifth Normal Form (5NF).

Let us first define the concept of PJNF from the viewpoint of the decomposition and then refine it later to a standard form.

Definition 1: A JD *[R1, R2, . . . , Rn] over a relation R is trivial if it is satisfied by every relation r(R).

The trivial JDs over R are JDs of the form *[R1, R2, . . . , Rn] where for some i the Ri = R.

Definition 2: A JD *[R1, R2, . . ., Rn] applies to a relation scheme R if R = R1 R2 …Rn.

Definition 3: Let R be a relation scheme having F as the set of FDs and JDs over R. R will be in project-join normal form (PJNF) if for every JD *[R1, R2, . . ., Rn] which can be derived by F that applies to R, the following holds:

- The JD is trivial, or
- Every Ri is a super key for R.

For a database scheme to be in project-join normal form, every relation *R* in this database scheme should be in project-join normal form with respect to *F*.

Let us explain the above with the help of an example.

Example: Consider a relational scheme R = *A B C D E G* having the set of dependencies F = {*[*A B C D, C D E, B D G*], *[*A B, B C D, A D*], A → *B C D E, B C → A G* }. The *R* as given above is not in PJNF. Why? The two alternate keys to *R* are *A* and *BC*, so please note that the *JD* *[*A B C D, C D E, B D G*], does not satisfy the condition "Every *Ri* is a super key for R" as the two components of this *JD* viz., *C D E* and *B D G*, does not satisfy the condition.

However, if we decompose the *R* as {*R1, R2, R3*}, where *R1 = A B C D, R2 = C D E,* and *R3 = B D G*, then it is in *PJNF* with respect to *F*. Please note that in the example, the *JD* *[*A B, B C D, A D*] is implied by F and applies to *R1*. Whereas, the *FDs* are trivial or have keys as the left side.

The definition of *PJNF* as given above is a weaker than the original definition of *PJNF* given by Fagin. The original definition ensures enforceability of dependencies by satisfying keys, in addition to elimination of redundancy. The final definition is:

Definition 4: Let *R* be a relation scheme having *F* as the set of *FDs* and *JDs* over *R. R* will be in project-join normal form (*PJNF*) if for every JD *[*R1, R2*, . . ., *Rn*] which can be derived by *F* that applies to *R*, is implied by the key *FDs* of *R*.

The following example demonstrates this definition.

Example: Consider a relation scheme *R = A B C* having the set of dependencies as *F* = {A→ B C, C → A B, *[A B, B C] }. Please note that the *R* is not in *PJNF*, although since *A B* and *B C* are the super keys of *R, R* satisfies the earlier definition of *PJNF*. But R does not satisfy the revised definition as given above.

Please note that since every multivalued dependency is also a join dependency, every PJNF schema is also in 4NF. Decomposing a relation scheme using the JDs that cause PJNF violations creates the PJNF scheme. PJNF may also be not dependency preserving.

☞ Check Your Progress 2

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

……

……

……

2) Decompose the following into 4NF

EMP

ENAME	PNAME	DNAME
Dev	X	Sanju
Dev	Y	Sainyam
Dev	X	Sainyam
Dev	Y	Sanju

……

……

……

3) Does the following relation satisfy MVDs with 4NF? Decompose the following relation into 5NF.

SUPPLY

SNAME	PARTNAME	PROJNAME
Dev	Bolt	X
Dev	Nut	Y
Sanju	Bolt	Y
Sainyam	Nut	Z
Sanju	Nail	X
Sanju	Bolt	X
Dev	Bolt	Y

……

……

……..

4) When is a set of functional dependencies F minimal?

……

……

……

……..

5) Give the proof of transitive rule $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.

……

……

……

……..

1.7 INCLUSION DEPENDENCIES AND TEMPLATE DEPENDENCIES

An **inclusion dependency** $R.X < S.Y$ between two sets of attributes – X of a relation schema R, and Y of a relation schema S – is defined as the following constraint:

If r and s are the relation state of R and S respectively at any specific time then:

$$\pi_X(r(R)) \subseteq \pi_Y(s(S))$$

The subset relationship does not necessarily have to be a proper subset. Please note that the sets of attributes on which the inclusion dependency is specified viz. *X* of *R* and *Y* of *S* above, must have the same number of attributes. In addition, the domains for each corresponding pair of attributes in *X* and *Y* should be compatible. The objectives of inclusion Dependencies are to formalise two important types of interrelation constraints that exist between the relations, thus cannot be expressed using FDs and MVDs that are:

1) Referential integrity constraints,

2) Class / subclass relationships.

The common rules for making inferences from defined inclusion dependencies (Inclusion Dependency Inference Rule – IDIR) are:

IDIR1 (reflexivity): $R.X < R.X$

IDIR2 (attribute correspondence): if $R.X < S.Y$

here

X = {*A1, A2, ..., An*} and
Y = {*B1, B2, ..., Bn*} and
Ai correspondence to *B*i for $1 \leq i \leq n$

IDIR3 (transitivity): If R.X < S.Y and S.Y < T.Z then R.X < T.Z

Template Dependencies[1]

The template dependencies are the more general and natural class of data dependencies that generalizes the concepts of JDs. A template dependency is representation of the statement that a relation is invariant under a certain tableau mapping. Therefore, it resembles a tableau. It consists of a number of hypothesis rows that define certain variables with a special row at the bottom, called the conclusion row. A relation r satisfies a template dependency, if and only if, a valuation (say ρ) that successfully maps the hypothesis rows to tuples in a relation r, finds a map for conclusion row to a tuple in r. (Please note that this is not the complete and formal definition). Let us explain this informal definition with the help of example.

Example 1: The Figure 6 shows a template dependency *T* over the scheme *A B C* having specific variables of A (a and a'), B and C. The hypothesis rows are *w1, w2, w3* and *w4.* The row w is the conclusion row. Relation *r* given in Figure 7 does not satisfy *T*, since the valuation ρ that maps hypothesis rows *w1* to *w4* using variable values a = 1, a' = 2, b = 3, b' = 4, c = 5, c' = 6, does not map conclusion row w to any tuple in r. To make the relation in Figure 7 to satisfy the template dependency given in

(http://www.dbis.informatik.hu-berlin.de/~freytag/Maier/C14.pdf)

Figure 6, we need to add a tuple that is equivalent to the conclusion row. Thus, we need to add a tuple t5 in figure 1.7 having the values of the variables a, b and c as 1,3,5 respectively. Please note the template dependencies are difficult to check in a large table.

T	(A	B	C)
wl	a	b	c’
w2	a’	b	c
w3	a	b’	c
w4	a’	b’	c
w	a	b	c

Figure 6: A sample template dependency

R	(A	B	C)
t1	l	3	6
t2	2	3	5
t3	l	4	5
t4	2	4	5

Figure 7: An example relation *r* for checking template dependency

Please note that although the template dependencies look like tableaux, but they are not exactly the same. We will discuss about this concept in more details later in this section. Let us now define the template dependency formally:

Definition: A template dependency (*TD*) on a relation scheme R is a pair T =(T,w) where T={w1,w2, ... wk } is a set of hypothesis rows on R, and w is a single conclusion row on R. A relation r(R) satisfies *TD T* if for every valuation ρ of T such that $\rho(T) \subseteq r$, ρ can be extended to show that $\rho(w) \in r$. A template dependency is trivial if every relation over R satisfies it.

The template dependencies are written as shown in *Figure 6.* The conclusion row is written at the bottom separated from the hypothesis rows. The variables are written using lowercase alphabets corresponding to possible attribute name. The conclusion row variables are normally have not primed or subscripted. The TDs almost look like tableau mappings turned upside down. A template dependency is different from a tableau in the following two ways:

1) A variable like (a, b, c etc.) in the conclusion row need not appear in any of the hypothesis row.
2) Variables may not be necessarily restricted to a single column.

Let us show both the points above with the help of an example each.

Example 2: Consider the TD *T* on scheme A B C in *Figure 8*. It is a valid TD expression; please note that the variable c is not appearing in the hypothesis rows where the variables are c’ and c”. This TD has the variable of conclusion row on A and B in the hypothesis rows, but not on C, therefore, is called A B-partial.

T (A	B	C)
a’	b	c”
a’	b	c’
a	b’	c”
a	b	c

Figure 8: A sample A B-partial TD*T*

A TD*T* on scheme R where every variable in the conclusion row appears in some hypothesis row is termed as *full*. Consider a TD having wl, w2, . . . , wk as the hypothesis rows and w as the conclusion row, a TD is called S-partial, where S is the set defined as: {S ∈ R | w(S) appears in one of w1, w2, . . . , wk). The TD is full if the S = R and strictly partial if S ≠ R.

Let us now define second difference, but to do so let us first define few more terms. A TD in which each variable appears in exactly one column is called a *typed* TD, but if some variable appears in multiple columns then it is called an *untyped* TD. The TDs shown in *Figures 6* and *8* are typed.

Example 3: *Figure 9* shows an untyped TD*T*. This TD assumes that the domain of A is same as that of domain of B, otherwise such TD will not make any sense.

T (A	B)
b	c
a	b
a	c

Figure 9: Untyped TD*T*

Let us now show the relationship of JD and MVD to the TD.

Example 4: Consider the MVD A →→ B over the relation scheme A B C is equivalent to the TD *T* in *Figure 10* . TD*T indicates* that if a relation has two tuples tl and t2 that agree on A, it must also have a tuple t3 such that t3 (A B) = t1(A B) and t3 (A C) = t2 (A C), which is just a way of stating that the relation satisfies A→→ B.

T (A	B	C)
a	b	c'
a	b'	c
a	b	c

Figure 10: A TD*T* for MVD

However, please note that not every TD corresponds to a JD. This can be ascertained from the fact that their can an infinite number of different TDs over a given relation scheme, whereas there is only a finite set of JDs over the same scheme. Therefore, some of the TDs must not be equivalent to any JD.

1.8 DOMAIN KEY NORMAL FORM (DKNF)

The Domain-Key Normal Form (DKNF) offers a complete solution to avoid the anomalies. Thus, it is an important Normal form. A set of relations that are in DKNF must be free of anomalies. The DKNF is based on the 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 keys.

Key declaration: Let *R* be a relation schema with $K \subseteq R$. A key *K* requires that *K* be a superkey for schema *R* such that $K \rightarrow R$. 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.

Domain declaration: Let *A* be an attribute and **dom** be a set of values. The domain declaration stated as $A \subseteq$ **dom** requires that the values of *A* in all the tuples of R be values from **dom**.

Constraint is a well defined rule that is to be uphold by any set of legal data of R.
General constraint: 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.

A general constraint need not be a functional, multivalued, or join dependency. For example, in the student's enrolment number the first two digit represents year. Assuming all the students are MCA students and the maximum duration of MCA is 6 years, in the year 2006, the valid students will have enrolment number that consists of 00 as the first two digits. Thus, the general constraint for such a case may be: "If the first two digit of *t*[enrolment *number*] is 00, then *t*[marks] are valid."

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

Valid student schema =(enrolment number, subject, marks)

Invalid student schema = (enrolment number, subject, marks)

Please note that the schema of valid account number requires that the enrolment number of the student begin with the 00. The resulting design is in DKNF.

Please note that the constraints that are time-dependent or relate to changes made in data values were excluded from the definition of DKNF. This implies that a time-dependent constraint (or other constraint on changes in value) may exist in a table and may fail to be a logical consequence of the definitions of keys and domains, yet the table may still be in DKNF.

How to convert a relation to DKNF? There is no such direct procedure for converting a table into one or more tables each of which is in DKNF. However, as a matter of practice, the effort to replace an arbitrary table by a set of single-theme tables may covert a set of tables to DKNF.

A result of DKNF is that all insertion and deletion anomalies are removed. DKNF represents an "ultimate" normal form because it allows constraints, rather than dependencies. DKNF allows efficient testing of the constraints. Of course, if a schema is not in DKNF, we may have to perform decomposition, but such decompositions are not always dependency-preserving. Thus, although DKNF is an aim of a database designer, it may not be implemented in a practical design.

1.9 MODELING TEMPORAL DATA

Let us first define the concept of a temporal database. Broadly speaking, temporal databases are those database applications that involve some aspect of time. That broad definition puts many database applications that use to record the history of database with respect to the time of updating. Such application where time is a very critical factor may include "medical database systems" where the medical history of a patient is to be recorded along with the timestamp. Similarly, for a railway reservation system the time of booking of trains is important to check, if anybody is booking for the train that s/he cannot board. Also the time of cancellation is important as on this basis refunds are calculated. Another example may be the library information system where the book issue and return system is based on time. Many

such systems follow the concept of time. Temporal data adds complexity in a database application and is sometimes overlooked, thus resulting in loss of valuable information.

In a temporal database system you need to model time keeping the following points in mind:

- You need to define database as a sequence of time based data in chronological order.
- You need to resolve events that happen at the same time.
- A reference point of time may be defined to find time relative to it.
- Sometimes a calendar is used.
- The SQL support for temporal data includes:
    - data types such as Date (dd,mm,yyyy,
    - TIME (hh:mm:ss), TIMESTAMP.
      which specifies a unique sequence number, based on time, to identify sequence of events/activities, INTERVAL (time durations) and PERIOD (period frame reference point).
    - You can also define the concept of valid time for a data entity for example an assignment may be valid till a particular time.

☞ Check Your Progress 3

1) Define Inclusion Dependencies.

……

……

……

2) What is the key idea behind DKNF?

……

……

……

1.10 SUMMARY

In this unit we studied about several normalisation levels. We first discussed enhanced ER tools like Inheritance, generalisation and specialisation. We then defined additional types of dependencies with normal forms. Multivalued dependencies which arise due to a combination of two or more multivalued attributes in the same relation are used to define 4NF. Join dependencies, which results in lossless multiway decomposition of a relation, led us to the definition of 5NF, which is also known as PJNF. We also defined inclusion dependencies used to specify referential integrity and class/subclass constraints and template dependencies, used to specify arbitrary types of constraints. This unit also included a brief discussion on DKNF. You may refer to the further readings for more details on these topics.

1.11 SOLUTIONS/ANSWERS

Check Your Progress 1

1) The EER diagrams are used to model advanced data model requiring inheritance and specialisation and generalisation.

2) The basic constraints used in EER diagrams are disjointness, overlapping and unions.

3) For disjointness and union constraints the chances are that we create separate tables for the subclasses and no table for super class. For overlapping constraint it is advisable to have a table of super class. For such cases the tables of subclasses will have only those attributes that are not common to super class except for the primary key.

Check Your Progress 2

1) An MVD is a constraint due to multi-valued attributes. A relational must have at least 3 attributes out of which two should be Multi-valued.

2)

EMP_PROJECTS

ENAME	PNAME
DeV	X
Dev	Y

EMP_DEPENDENTS

ENAME	DNAME
Dev	Sanju
Dev	Sainyam

3)

No,

R1

SNAME	PARTNAME
Dev	Bolt
Dev	Nut
Sanju	Bolt
Sainyam	Nut
Sanju	Nail

R2

SNAME	PROJNAME
Dev	X
Dev	Y
Sanju	Y
Sainyam	Z
Sanju	X

R3

PARTNAME	PROJNAME
Bolt	X
Nut	Y
Bolt	Y
Nut	Z
Nail	X

4) A set of functional dependencies F is minimal if it satisfies the following conditions:

- Every dependency in F has a single attribute for its right-hand side.
- We cannot replace any dependency $X \rightarrow A$ in F with Dependency $Y \rightarrow A$, where Y is a proper subset of X and still have a set of dependencies that is

equivalent to *F*.

- We cannot remove any dependency from F and still have a set of dependencies that is equivalent to F.

5) 1. $X \rightarrow Y$ and 2. $Y \rightarrow Z$ both holds in a relation r. Then for any two tuples t1 and t2 in r such that t1[X] = t2[X], we must have 3. t1[Y] = t2[y], from assumption 1; hence we must also have 4. t1[Z] = t2[Z], from 3 and assumption 2; hence $X \rightarrow Z$ must hold in r.

Check Your Progress 3

1) An inclusion dependency R.X < S.Y between two sets of attributes – X of a relation schema R, and Y of a relation schema S – is defined as the following constraint:

   If r and s are the relation state of R and S respectively at any specific time then:

   $\prod_X(r(R)) \subseteq \prod_Y(s(S))$

   The subset relationship does not necessarily have to be a proper subset. Please note that the sets of attributes on which the inclusion dependency is specified viz., *X* of *R* and *Y* of *S* above, must have the same number of attributes. In addition, the domains for each corresponding pair of attributes in *X* and *Y* should be compatible.

2) To specify the "ultimate normal form".

1.12 FURTHER READINGS

1) *"Fundamentals of Database System"*, Elmasri, Ramez and Navathe Shamkant B., Forth Edition, Pearson Education, India, 2004.

2) *"Database System Concepts"*, Silberschatz A., Korth Henry F., S.Sudarshan, Fifth Edition, McGraw Hill, 2006.

3) Some very useful websites:

   http://www.dbis.informatik.hu-berlin.de/~freytag/Maier/C14.pdf
   http://wwwdb.informatik.uni-rostock.de/Lehre/Vorlesungen/MAIER/C07.pdf
   http://codex.cs.yale.edu/avi/db-book/online-dir/c.pdf
   http://www-staff.it.uts.edu.au/~zhangsc/scpaper/AJITzzqin.pdf