UNIT 3 OVERVIEW OF NORMALISATION

Structure

3.0 INTRODUCTION

We know that relations that form the database must satisfy some properties, for example, relations have no duplicate tuples, tuples have no ordering associated with them, and each element in the relation is atomic. Relations that satisfy these basic requirements may still have some undesirable properties, for example, data redundancy and update anomalies. We illustrate these properties and study how relations may be transformed or decomposed (or normalised) to eliminate them. Most such undesirable properties do not arise if the database modeling has been carried out very carefully using some technique like the Entity-Relationship Model that we have discussed but it is still important to understand the techniques in this chapter to check the model that has been obtained and ensure that no mistakes have been made in modeling.

The central concept in these discussions is the notion of functional dependency, which depends on the semantics of the data and which deals with what information in a relation is dependent on what other information in the relation.

This unit continues our discussion of design issues in relational databases. In general, the goal of a relational-database design is to generate a set of relational schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. One approach is to design schemas that are in an appropriate normal form. Our prime objective in this unit is to define the concept of functional dependence more precisely and then define normal forms using functional dependencies and using other types of data dependencies.

3.1 OBJECTIVES

At the end of this chapter, you should be able to:

- Define the concept of functional dependency;
- Discuss how to reason with the dependencies;
- Show how to use the dependencies information to decompose relations whenever necessary to obtain relations that have the desirable properties that we want without loosing any of the information in the original relations.

3.2 REDUNDANCY AND ASSOCIATED PROBLEMS

Let us consider the following student relation.

Sno	Sname	Address	Cno	Cname	Instructor	office
1123	Rahul	D-27, Dewas	CS -01	Computer Organisation	Ashish Kumar	102
1123	Rahul	D-27, Dewas	CS -03	COBOL	Anurag Sharma	105
1123	Rahul	D-27, Dewas	CS-06	DBMS	Preeti Anand	103
1134	Aparna	B-III, Gurgaon	CS-06	DBMS	Preeti Anand	103

The above table satisfies the properties of a relation and is said to be in first normal form (or 1NF). Conceptually it is convenient to have all the information in one relation since it is then likely to be easier to query the database. But the above relation has the following undesirable features:

Repetition of information: A lot of information is being repeated. Student name, address, course name, instructor name and office number are being repeated often. Every time we wish to insert a student enrolment, say, in CS-06 we must insert the name of the course CS-06 as well as the name and office number of its instructor. Also every time we insert a new enrolment for, say Rahul, we must repeat his name and address. Repetition of information results in wastage of storage as well as other problems.

1. **Update Anomalies:** Redundant information not only wastes storage but also makes updates more difficult since, for example, changing the name of the instructor of CS-06 would require that all tuples containing CS-06 enrolment information be updated. If for some reason, all tuples are not updated, we might have a database that gives two names of instructor for subject CS-06. This difficulty is called the update anomaly.

2. **Insertional Anomalies:** Inability to represent certain information– Let the primary key of the above relation be (sno, cno). Any new tuple to be inserted in the relation must have a value for the primary key since existential integrity requires that a key may not be totally or partially NULL. However, if one wanted to insert the number and name of a new course in the database, it would not be possible until a student enrols in the course and we are able to insert values of sno and cno. Similarly information about a new student cannot be inserted in the database until the student enrols in a subject. These difficulties 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 we delete the tuple corresponding to student 1123 doing CS-03, we will loose relevant information about course CS-03 (viz., course name, instructor, office number) if the student 1123 was the only student enrolled in that course. Similarly deletion of course CS-06 from the database may remove all information about the student named Aparna. This is called deletion anomalies.

The above problems arise primarily because the **student** relation has information about students as well as subjects. One solution to deal with the problems is to decompose the relation into two or more smaller relations.

Decomposition may provide further benefits, for example, in a distributed database different relations may be stored at different sites if necessary. Of course, decomposition does increase the cost of query processing since the decomposed relations will need to be joined, sometime frequently.

The above relation may be easily decomposed into three relations to remove most of the above undesirable properties:

S(sno,sname,address)

C(cno,cname,instructor,office)

SC(sno, cno)

Such decomposition is made using the principles of normalisation and is essential if we wish to overcome undesirable anomalies.

As noted earlier, normalisation often has an adverse effect on performance. Data which could have been retrieved from one relation before normalisation may require several relations to be joined after normalisation. Normalisation does however lead to more efficient updates since an update that may have required several tuples to be updated before normalisation could well need only one tuple to be updated after normalisation. It also results in overall size reduction of a database, and is MOST desirable for any database design.

Although in the above case we are able to look at the original relation and propose a suitable decomposition that eliminates the anomalies that we have discussed, in general this approach is not possible. A relation may have one hundred or more attributes and it is then almost impossible for a person to conceptualise all the information and suggest a suitable decomposition to overcome the problems. We therefore need an algorithmic approach to finding if there are problems in a proposed database design and how to eliminate them if they exist.

There are several stages of the normalisation process. These are called the first normal form (1NF), the second normal form (2NF), the third normal form (3NF), Boyce-Codd normal form (BCNF), the fourth normal form (4NF) and the fifth normal form (5NF).

For all practical purposes, 3NF or the BCNF are quite adequate since they remove the anomalies discussed above 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 this may result in a decision not to normalise.

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). That is, non-key attributes in each relation must provide a fact about the entity or relationship that is being identified by the key. Again, 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 3NF.

It should be noted that decomposition of relations has to be always based on principles that ensure that the original relation may be reconstructed from the decomposed relations if and when necessary. If we are able to reduce redundancy and not loose any information, it implies that all that redundant information can be derived given the other information in the database. Therefore information that has been removed must be related or dependent on other information that still exists in the database. That is why the concept of redundancy is important. Careless decomposition of a relation can result in loss of information. We will discuss this in detail later.

3.3 ROLE OF NORMALIZATION

In the E-R model, a conceptual schema using an entity- relationship diagram is built and then mapped to a set of relations. This technique ensures that each entity has information about only one thing and once the conceptual schema is mapped into a set of relations, each relation would have information about only one thing. The relations thus obtained would normally not suffer from any of the anomalies that have been discussed in the last section. It can be shown that if the entity-relationship is built using the guidelines that were presented in last unit, the resulting relations are in 3NF.

Of course, mistakes can often be made in database modeling specially when the database is large and complex or one may, for some reasons, carry out database schema design using techniques other than a modeling technique like the entity- relationship model.

For example, one could collect all the information that an enterprise possesses and build one giant relation (often called the universal relation) to hold it. This bottom-up approach is likely to lead to a relation that is likely to suffer from all the problems that we have discussed in the last section. For example, the relation is highly likely to have redundant information and update, deletion and insertion anomalies. Normalisation of such large relation will then be essential to avoid (or at least minimise) these problems.

To summarise, normalisation plays only some basic role in database schema design if a top-down modeling approach like the entity- relationship approach is used. Normalisation, however, plays a major role when the bottom-up approach is being used. Normalisation is then essential to build appropriate relations to hold the information of the enterprise.

Now to define the normal forms more formally, we first need to define the concept of functional dependence.

3.4 SINGLE-VALUED 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 multivalued. Name of a person or his date of birth are single-valued facts; qualifications of a person or subjects that an instructor teaches are multivalued facts. We first deal only with single-valued facts and discuss the concept of functional dependency.

The concept of some information being dependent on other is quite basic; for example, at any time in a database, a student number would always determine the student name (a person may change his/her name but at any one time the person's name can be determined by the student number) and a subject with a given subject number would always have the same name. We formalise this concept.

Functional Dependency

We can say that the notion of functional dependency is a generalisation of the notion of key.
Superkey: Let R be a relation schema. A subset K of R is a superkey of R if, in any legal relation r(R), for all pairs t_1 and t_2 of tuples in r such that $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$. That is, no two tuples in any legal relation r(R) may have the same value on attribute set K.

The notion of functional dependency generalises the notion of superkey. We can define functional dependency as follows:

Definition: Let $\alpha \subseteq$ R and $\beta \subseteq$ R. The functional dependency $\alpha \rightarrow \beta$ holds on R if, in any legal relation r(R), for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] \neq t_2[\alpha]$, it is also the case that $t_1[\beta] \neq t_2[\beta]$.

We also often express this idea by saying that "α determines β", or that "β is a function of α", or that "α functionally governs β". Often, the notions of functionality and functional dependency are expressed briefly by the statement, "If α, then β". It is important to note that the value β must be unique for a given value of α, i.e., any given value of α must imply just one and only one value of β, in order for the relationship to qualify for the name "function". (However, this does not necessarily prevent different values of α from implying the same value of β.)

Consider a relation R that has two attributes A and B. The attribute B of the relation is functionally dependent on the attribute A if and only if for each value of A no more than one value of B is associated. In other words, the value of attribute A uniquely determines the value of B and if there were several tuples that had the same value of A then all these tuples will have an identical value of attribute B. That is, if $\mathbf{t_1}$ and $\mathbf{t_2}$ are two tuples in the relation R and $\mathbf{t_1(A) = t_2(A)}$ then we must have $\mathbf{t_1(B) = t_2(B)}$.

A and B need not be single attributes. They could be any subsets of the attributes of a relation R (possibly single attributes). We may then write–

R.A → R.B

if B is functionally dependent on A (or A functionally determines B). Note that functional dependency does not imply a one-to-one or one to many relationship between A and B although a one-to-one or many-to-one relationship may exist between A and B.

A simple example of the above functional dependency is when A is a primary key of an entity (e.g., student number) and A is some single-valued property or attribute of the entity (e.g., date of birth). **A → B** then must always hold. (why?)

Functional dependencies also arise in relationships. Let C be the primary key of an entity and D be the primary key of another entity. Let the two entities have a relationship. If the relationship is one-to-one, we must have **C → D** and **D → C** . If the relationship is many-to-one, we would have **C→ D** but not **D → C** . For many-to-many relationships, no functional dependencies hold. For example, if C is student number and D is subject number, there is no functional dependency between them. If however, we were storing marks and grades in the database as well, we would have the functional dependency–

(student number , subject number) → marks

and we might have the functional dependency

marks → grades

The second functional dependency above assumes that the grades are dependent only on the marks. This may sometime not be true since the instructor may decide to take other considerations into account in assigning grades, for example, the class average mark.

In the student database that we have discussed earlier, we have the following functional dependencies:

sno → sname
sno → address
cno → cname
cno → instructor
instructor → office

These functional dependencies imply that there can be only one name for each sno, only one address for each student and only one subject name for each cno. It is of course possible that several students may have the same name and several students may live at the same address. If we consider **cno → instructor**, 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 the above example, one may be wondering if the following FDs hold

sname → sno
cname → cno

Certainly there is nothing in the instance of the example database presented above that contradicts the above functional dependencies. However, whether above FDs hold or not would depend on whether the university or college whose database we are considering allows duplicate student names and subject names. If it was the enterprise policy to have unique subject names then **cname → cno** holds. If duplicate student names are possible, and one would think there always is the possibility of two students having exactly the same name, then **sname → sno** does not hold.

Functional dependencies arise from the nature of the real world that the database models. Often A and B are facts about an entity where A might be some identifier for the entity and B some characteristic. Functional dependencies cannot be automatically determined by studying one or more instances of a database. They can be determined only by a careful study of the real world and a clear understanding of what each attribute means.

We have noted above that the definition of functional dependency does not require that A and B be single attributes. In fact, A and B may be collections of attributes. For example,

(sno , cno) → (marks , date)

When dealing with a collection of attributes, the concept of full functional dependence is an important one. Let A and B be distinct collections of attributes from a relation R end let **R.A→R.B**. B is then fully functionally dependent on A if B is not functionally dependent on any subset of A. The above example of students and subjects would show full functional dependence if mark and date are not functionally dependent on either student number (sno) or subject number (cno) alone. The implies that we are assuming that a student may have more than one subjects and a subject would be taken by many different students. Furthermore, it has been assumed that there is at most one enrolment of each student in the same subject.

The above example illustrates full functional dependence. However the following dependence

(sno , cno) → instructor

is not full functional dependence because **cno → instructor** holds.

As noted earlier, the concept of functional dependency is related to the concept of candidate key of a relation since a candidate key of a relation is an identifier which uniquely identifies a tuple and therefore determines the values of all other attributes in the relation. Therefore any subset X of the attributes of a relation R that satisfies the property that all remaining attributes of the relation are functionally dependent on it (that is, on X), then X is candidate key as long as no attribute can be removed from X and still satisfy the property of functional dependence. In the example above, the attributes (sno, cno) form a candidate key (and the only one) since they functionally determine all the remaining attributes.

Functional dependence is an important concept and a large body of formal theory has been developed about it. It is also associated with the concept of closure that helps us derive all functional dependencies that are implied by a given set of dependencies. A complete set of functional dependencies can be obtained, by studying the constraints in a relation relating to various attributes.

3.5 SINGLE VALUED NORMALISATIONS

Initially Codd (1972) presented three normal forms (1NF, 2NF, and 3NF) all 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 multivalued and join dependencies and were proposed later.

3.5.1 The First Normal Form (1NF)

A table satisfying the properties of a relation is said to be in first normal form. As discussed in the previous unit, a relation cannot have multivalued or composite attributes. This is what the 1NF needs to resolve.

A relation is in 1NF if and only if all underlying domains contain atomic values only.

Definition: A table (relation) is in 1NF if

1. **There are no duplicated rows in the table.**
2. **Each cell is single-valued (i.e., there are no repeating groups or arrays).**
3. **Entries in a column (attribute, field) are of the same kind.**

Notes: The order of the rows is immaterial; the order of the columns is immaterial.

The requirement that there be no duplicated rows in the table means that the table has a key (although the key might be made up of more than one column--even, possibly, of all the columns).

The first normal form deals only with the basic structure of the relation and does not resolve the problems of redundant information or the anomalies discussed earlier. All relations discussed in these notes are in 1NF.

For example, consider the following example relation–

student (sno, sname, dob)

Add some other attributes so it has anomalies and is not in 2NF.

The attribute dob is the date of birth and the primary key of the relation is sno with the functional dependencies **sno → sname** and **sno → dob** . The relation is in 1NF as long as dob is considered an atomic value and not consisting of three components (day, month, year). The above relation of course suffers from all the anomalies that we have discussed earlier and needs to be normalised further.

3.5.2 The Second Normal Form (2NF)

The second normal form attempts to deal with the problems that are identified with the relation above that is in 1NF. The aim of second normal form is to ensure that all information in one relation is only about one thing.

Definition: A relation is in 2NF if it is in 1NF and every non-key attribute is fully dependent on each candidate key of the relation.

Note: Since a partial dependency occurs when a non-key attribute is dependent on only a part of the (composite) key, the definition of 2NF is sometimes phrased as, "A table is in 2NF if it is in 1NF and if it has no partial dependencies."

To understand the above definition of 2NF we need to define the concept of key attributes. Each attribute of a relation that participates in at least one candidate key of the relation is a key attribute of the relation. All other attributes are called non-key.

The concept of 2NF requires that all attributes that are not part of a candidate key be fully dependent on each candidate key. Let us consider the relation:

student (sno, sname, cno, cname)

and the constraints that a course number is given to only one course name; the functional dependencies are:

sno → sname

cno → cname

and assume that **(sno, cno)** is the only candidate key (and therefore the primary key), the relation is not in 2NF since **sname** and **cname** are not fully dependent on the key. The above relation suffers from the same anomalies and repetition of information as discussed above since sname and cname will be repeated. To resolve these difficulties we could remove those attributes from the relation that are not fully dependent on the candidate keys of the relations. Therefore we decompose the relation into the following projections of the original relation:

S1 (sno, sname)

S2 (cno, cname)

SC (sno, cno)

We may recover the original relation by taking the natural join of the three relations. If, however, we assume that sname and cname are unique and therefore we have the following candidate keys:

(sno, cno)

(sno, cname)

(sname, cno)

(sname, cname)

The original student relation in that case is in 2NF since the relation has no non-key attributes. The relation still has the same problems as before but it then does satisfy the requirements of 2NF. Higher level normalisation is needed to resolve such problems with relations that are in 2NF and further normalisation will result in decomposition of such relations.

3.5.3 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 that appear in the relation that was not in 2NF, not all anomalies are removed and further normalisation is sometime needed to ensure further removal of anomalies. These anomalies arise because a 2NF relation may have attributes that are not directly dependent on the candidate key.

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

To understand the third normal form, we need to define 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**. As noted earlier, this dependence **A → C** is transitive, which exist because **A → B** and **B → C** exist.

The 3NF differs from the 2NF in that all non-key attributes in 3NF are required to be directly dependent on each candidate key of the relation. The 3NF therefore insists, in the words of Kent (1983) that all facts in the relation are about the key (or the thing that the key identifies), the whole key and nothing but the key. If some attributes are dependent on the keys transitively then that is an indication that those attributes provide information not about the key but about a non-key attribute. So the information is not directly about the key, although it obviously is related to the key.

Consider the following relation:

subject (cno, cname, instructor, office)

Assume that **cname** is not unique and therefore **cno** is the only candidate key. The following functional dependencies exist–

cno → cname

cno → instructor (One course is being taught by only one instructor)

instructor → office (An instructor has only one office place)

We can derive **cno → office** from the above functional dependencies and, therefore, the above relation is in 2NF. The relation is however not in 3NF since office is not directly dependent on cno. This transitive dependence is an indication that the relation has information about more than one thing (viz., course and instructor) and should therefore be decomposed. The primary difficulty with the above relation is that an instructor might be responsible for several subjects and, therefore, his office address may need to be repeated many times. This leads to all the problems that we identified at the beginning of this unit. To overcome these difficulties we need to decompose the above relation in the following two relations–

s (cno, cname, instructor)

ins (instructor, office)

Relation **s** is now in 3NF and so is relation **ins**.

An alternate decomposition of the relation subject is possible–

s(cno, cname)

inst(instructor, office)

si(cno, instructor)

The decomposition into three relations is not necessary, since the original relation is based on the assumption of one instructor for each course.

The 3NF is usually quite adequate for most relational database designs. There are however some situations, for example the relation **student (sno, sname, cno, cname)** discussed in 2NF above, where 3NF may not eliminate all the redundancies and inconsistencies. The problem with the relation **student (sno, sname, cno, cname)** is because of the redundant information in the candidate keys. These are resolved by further normalisation using the BCNF.

3.5.4 Boyce-Codd Normal Form (BCNF)

The relation **student (sno, sname, cno, cname)** has all attributes participating in candidate keys since all the attributes are assumed to be unique. We, therefore, had the following candidate keys:

(sno, cno)

(sno, cname)

(sname, cno)

(sname, cname)

Since the relation has no non-key attributes, the relation is in 2NF and also in 3NF, in spite of the relation suffering the problems that we discussed at the beginning of this unit.

The difficulty in this relation is being caused by dependence within the candidate keys. The second and third normal forms assume that all attributes not part of the candidate keys depend on the candidate keys but does not deal with dependencies within the keys. BCNF deals with such dependencies.

Definition: A relation R is said to be in BCNF if $X \rightarrow A$ holds in R, and A is not in X, then X is a candidate key for R. In other words, a relation is in BCNF if it is in 3NF and if every determinant (left hand side of a functional dependency) is a candidate key.

It should be noted that most relations that are in 3NF are also in BCNF. Infrequently, a 3NF relation is not in BCNF and this happens only if,

(a) The candidate keys in the relation are composite keys (that is, they are not single attributes),

(b) There is more than one candidate key in the relation, and

(c) The keys are not disjoint, that is, some attributes in the keys are common.

The BCNF differs from the 3NF only when there are more than one candidate keys and the keys are composite and overlapping. Consider for example, the relationship–

enrol (sno, sname, cno, cname, date-enrolled)

Let us assume that the relation has the following candidate keys

(sno, cno)

(sno, cname)

(sname, cno)

(sname, cname)

(we have assumed sname and cname are unique identifiers). The relation is in 3NF but not in BCNF because there are dependencies.

sno → sname

cno → cname

where attributes that are part of a candidate key are dependent on part of another candidate key. Such dependencies indicate that although the relation is about some entity or association that is identified by the candidate keys e.g., (sno, cno), there are attributes that are not about the whole thing that the keys identify. For example, the above relation is about an association (enrolment) between students and subjects and therefore the relation needs to include only one identifier to identify students and one identifier to identify subjects. Providing two identifiers about students (sno, sname) and two keys about subjects (cno, cname) means that some information about students and subjects that is not needed is being provided. This provision of information will result in repetition of information and the anomalies that we discussed at the beginning of this unit. If we wish to include further information about students and courses in the database, it should not be done by putting the information in the present relation but by creating new relations that represent information about entities student and subject.

These difficulties may be overcome by decomposing the above relation in the following three relations:

(sno, sname)

(cno, cname)

(sno, cno, date-of-enrolment)

We now have a relation that only has information about students, another only about subjects and the third contains the details about registration.

3.6 DESIRABLE PROPERTIES OF DECOMPOSITIONS

So far our approach has consisted of looking at individual relations and checking if they belong to 2NF, 3NF or BCNF. If a relation was not in the normal form that was being checked for and we wished the relation to be normalised to that normal form so that some of the anomalies can be eliminated, it was necessary to decompose the relation in two or more relations. The process of decomposition of a relation R into a set of relations **R1**$_1$ **R2**$_1$..... **Rn**$_1$ was based on identifying different components and using that as a basis of decomposition. The decomposed relations $\mathbf{R1_1}$ $\mathbf{R2_1}$..... $\mathbf{Rn_1}$ are projections of R and are of course not disjoint otherwise the glue holding the information together would be lost. Decomposing relations in this way based on a recognise and split method is not a particularly sound approach since we do not even have a basis to determine that the original relation can be constructed if necessary from the decomposed relations. We now discuss desirable properties of good decomposition and identify difficulties that may arise if the decomposition is done without adequate care. The next section will discuss how such decomposition may be derived given the FDs.

Desirable properties of a decomposition are:

- Attribute preservation;
- Lossless-join decomposition;
- Dependency preservation;
- Lack of redundancy.

3.6.1 Attribute Preservation

This is a simple and an obvious requirement that involves preserving all the attributes that were there in the relation that is being decomposed.

3.6.2 Lossless-Join Decomposition

In this unit, so far we have normalised a number of relations by decomposing them. We decomposed a relation intuitively. We need a better basis for deciding decompositions since intuition may not always be correct. We illustrate how a careless decomposition may lead to problems including loss of information.

Consider the following relation.

enrol (sno, cno, date-enrolled, room-No., instructor)

Suppose we decompose the above relation into two relations enrol1 and enrol2 as follows:

enrol1 (sno, cno, date-enrolled)

enrol2 (date-enrolled, room-No., instructor)

There are problems with this decomposition but we wish to focus on one aspect at the moment. Let an instance of the relation enrol be–

Sno	**cno**	**Date-enrolled**	**Room-No.**	**Instructor**
1123	CS-01	20-06-1999	**1**	Akshay Kumar
1123	CS-02	26-09-1999	**2**	Anurag Sharma
1259	CS-01	26-09-1998	**1**	Preeti Anand
1134	CS-05	30-10-1999	**5**	Preeti Anand
2223	CS-06	05-02-1998	**6**	Shashi Bhushan

and let the decomposed relations enrol1 and enrol2 be–

Sno	cno	Date-enrolled
1123	CS-01	20-06-1999
1123	CS-02	26-09-1999
1259	CS-01	26-09-1998
1134	CS-05	30-10-1999
2223	CS-06	05-02-1998

Date-enrolled	Room-No.	Instructor
20-06-1999	1	Akshay Kumar
26-09-1999	2	Anurag Sharma
26-09-1998	1	Preeti Anand
30-10-1999	5	Preeti Anand
05-02-1998	6	Shashi Bhushan

All the information that was in the relation enrol appears to be still available in enrol1 and enrol2 but this is not so. Suppose, we wanted to retrieve the student numbers of all students taking a course from Preeti Anand, we would need to join enrol1 and enrol2. The join would have 11 tuples as follows:

Sno	cno	Date-enrolled	Room-No.	Instructor
1123	CS-01	20-06-1999	1	Akshay Kumar
1123	CS-02	20-06-1999	1	Anurag Sharma
1123	CS-01	20-06-1999	1	Preeti Anand
1123	CS-05	20-06-1999	5	Preeti Anand
1123	CS-06	20-06-1999	6	Shashi Bhushan

(add further tuples ...)

The join contains a number of spurious tuples that were not in the original relation Enrol. Because of these additional tuples, we have lost the information about which students take courses from Preeti Anand. (Yes, we have more tuples but less information because we are unable to say with certainty who is taking courses from Preeti Anand). Such decompositions are called **lossy decompositions**. A nonloss or lossless decomposition is that which 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?).

We need to analyse why some decompositions are lossy. The common attribute in above decompositions was Date-enrolled. 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 is not unique, the relationship information is not preserved. If each tuple had a unique value of Date-enrolled, the problem of losing information would not have existed. The problem arises because several enrolments may take place on the same date.

A decomposition of a relation R into relations $\mathbf{R1}_1$ $\mathbf{R2}_1$..... $\mathbf{Rn}_1$ is called a lossless-join decomposition (with respect to FDs F) if the relation R is always the natural join of the relations $\mathbf{R1}_1$ $\mathbf{R2}_1$..... $\mathbf{Rn}_1$. It should be noted that natural join is the only way to recover the relation from the decomposed relations. There is no other set of operators that can recover the relation if the join cannot.

It is not difficult to test whether a given decomposition is lossless-join given a set of functional dependencies F. We consider the simple case of a relation R being decomposed into R_1 and R_2. If the decomposition is lossless-join, then one of the following two conditions must hold

$\mathbf{R_1 \cap R_2 \rightarrow R_1 - R_2}$

$\mathbf{R_1 \cap R_2 \rightarrow R_2 - R_1}$

That is, the common attributes in $\mathbf{R_1}$ and $\mathbf{R_2}$ must include a candidate key of either $\mathbf{R_1}$ or $\mathbf{R_2}$.

3.6.3 Dependency Preservation

It is clear that a decomposition must be lossless so that we 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 and if holds than we know that the two (sets) attributes are closely related and it would be useful if both attributes appeared in the same relation so that the dependency can be checked easily.

Let us consider a relation R(A, B, C, D) that has the dependencies F that include the following:

$\mathbf{A \rightarrow B}$

$\mathbf{A \rightarrow C}$

etc.

If we decompose the above relation into R1(A, B) and R2(B, C, D) the dependency $\mathbf{A \rightarrow C}$ cannot be checked (or preserved) by looking at only one relation. It is desirable that decompositions be such that each dependency in F may be checked by looking at only one relation and that no joins need be computed for checking dependencies. In some cases, it may not be possible to preserve each and every dependency in F but as long as the dependencies that are preserved are equivalent to F, it should be sufficient.

Let F be the dependencies on a relation R which is decomposed in relations $\mathbf{R1_1}$ $\mathbf{R2_{1.....}}$ $\mathbf{Rn_1}$. We can partition the dependencies given by F such that $\mathbf{F1_1}$ $\mathbf{F2_{1.....}}$ $\mathbf{Fn_1}$. **Fn** are dependencies that only involve attributes from decomposed relations respectively. If the union of dependencies imply all the dependencies in F, then we say that the decomposition has preserved dependencies, otherwise not.

If the decomposition does not preserve the dependencies F, then the decomposed relations may contain relations that do not satisfy F or the updates to the decomposed relations may require a join to check that the constraints implied by the dependencies still hold.

Consider the following relation

sub(sno, instructor, office)

We may wish to decompose the above relation to remove the transitive dependency of office on sno. A possible decomposition is

S1(sno, instructor)

S2(sno, office)

The relations are now in 3NF but the dependency **instructor → officer** cannot be verified by looking at one relation; a join of S1 and S2 is needed. In the above decomposition, it is quite possible to have more than one office number for one instructor although the functional dependency **instructor → officer** does not allow it. Thus, a decomposition may be lossless, yet not dependency preserving. The correct decomposition for the above relation will be **S1(sno, instructor)** and **S2(instructor, office)**

3.6.4 Lack of Redundancy

We have discussed the problems of repetition of information in a database. Such repetition should be avoided as much as possible.

Lossless-join, dependency preservation and lack of redundancy not always possible with BCNF. Lossless-join, dependency preservation and lack of redundancy is always possible with 3NF.

3.6.5 Deriving BCNF

Should we also include deriving 3NF?

Given a set of dependencies F, we may decompose a given relation into a set of relations that are in BCNF using the following algorithm

So far we have considered the "recognise and split" method of normalisation. We now discuss Bernstein's algorithm. The algorithm consists of

1) Find out the facts about the real world.
2) Reduce the list of functional relationships.
3) Find the keys.
4) Combine related facts.

Once we have obtained relations by using the above approach we need to check that they are indeed in BCNF. If there is any relation R that has a dependency **A → B** and A is not a key, the relation violates the conditions of BCNF and may be decomposed in AB and R - A. The relation AB is now in BCNF and we can now check if R - A is also in BCNF. If not, we can apply the above procedure again until all the relations are in fact in BCNF.

3.7 MULTIVALUED DEPENDENCIES

With modelling using the E-R Modelling technique, we noted difficulties that can arise when an entity has multivalue attributes. It was because 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 that we wish to represent. This results in many 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). Of course the other option suggested was to represent this multivalue information in a separate relation. The situation of course 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 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.

Consider the following relation that represents an entity employee that has one mutlivalued attribute proj:

emp (e#, dept, salary, proj)

We have so far considered normalization based on functional dependencies; dependencies that apply only to single-valued facts. For example, **e# → dept** implies only one dept value for each value of e#. Not all information in a database is single-valued, for example, proj 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 projects that an employee is working on, **e# → proj** is not a functional dependency.

So far we have dealt with multivalued facts about an entity by having a separate relation for that multivalue attribute and then inserting a tuple for each value of that fact. This resulted in composite keys since the multivalued fact must form part of the key. In none of our examples so far have we dealt with an entity having more than one multivalued attribute in one relation. We do so now.

The fourth normal form deals with multivalued dependencies. Before discussing the 4NF, we discuss the following example to illustrate the concept of multivalued dependency.

programmer (emp_name, qualifications, languages)

The above relation includes two multivalued attributes of entity programmer; qualifications and languages. There are no functional dependencies.

The attributes qualifications and languages are assumed **independent** of each other. If we were to consider qualifications and languages separate entities, we would have two relationships (one between emoloyees and qualifications and the other between employees and programming languages). Both the above one programmer could have several qualifications and may know several programming languages. Also one qualification may be obtained by several programmers and one programming language may be known to many programmers.

The above relation is therefore in 3NF (even in BCNF) but it still has some disadvantages. Suppose a programmer has several qualifications (B.Sc, Dip. Comp. Sc., etc.) and is proficient in several programming languages; how should this information be represented? There are several possibilities.

Emp_name	**qualifications**	**languages**
Rahul	B. Sc.	FORTRAN
Rahul	B. Sc.	COBOL
Rahul	B. Sc.	PASCAL
Rahul	Dip. CS	FORTRAN
Rahul	Dip. CS	COBOL
Rahul	Dip. CS	PASCAL

Emp_name	**qualifications**	**languages**
Rahul	B. Sc.	NULL
Rahul	Dip. CS	NULL
Rahul	NULL	FORTRAN
Rahul	NULL	COBOL
Rahul	NULL	PASCAL

Emp_name	qualifications	languages
Rahul	B. Sc.	FORTRAN
Rahul	Dip. CS	COBOL
Rahul	NULL	PASCAL

Other variations are possible (we remind the reader that there is no relationship between qualifications and programming languages). All these variations have some disadvantages. If the information is repeated we face the same problems of repeated information and anomalies as we did when second or third normal form conditions are violated. If there is no repetition, there are still some difficulties with search, insertions and deletions. For example, the role of NULL values in the above relations is confusing. Also the candidate key in the above relations is (emp name, qualifications, language) and existential integrity requires that no NULLs be specified.
These problems may be overcome by decomposing a relation like the one above as follows:

Emp_name	qualifications
Rahul	B. Sc.
Rahul	Dip. CS

Emp_name	languages
Rahul	FORTRAN
Rahul	COBOL
Rahul	PASCAL

The basis of the above decomposition is the concept of multivalued dependency (MVD). Functional dependency relates one value of A to one value of B while multivalued dependency **A ->> B** defines a relationship in which a set of values of attribute B are determined by a single value of A.

The concept of multivalued dependencies was developed to provide a basis for decomposition of relations like the one above. Therefore, if a relation like enrolment(sno, subject#) has a relationship between sno and subject# in which sno uniquely determines the values of subject#, the dependence of subject# on sno is called a trivial MVD since the relation enrolment cannot be decomposed any further. More formally, a MVD **X ->> 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. Therefore, a relation having non-trivial MVDs must have at least three attributes; two of them multivalued. Non-trivial MVDs result in the relation having some constraints on it since all possible combinations of the multivalue attributes are then required to be in the relation.

Let us now define the concept of multivalued dependency.

Definition: The multivalued dependency X ->> 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 attributes 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.

In the example above, if there was some dependence between the attributes qualifications and language, for example, perhaps, the language was related to the qualifications (perhaps the qualification was a training certificate in a particular language), and then the relation would not have MVD and could not be decomposed into two relations as above. In the above situation whenever **X ->> Y** holds, so does **X ->> Z** since the role of the attributes Y and Z is symmetrical.

Consider two different situations.

a) Z is a single valued attribute. In this situation, we deal with R(X, Y, Z) as before by entering several tuples about each entity.

b) Z is multivalued.

Now, more formally, **X ->> 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 and therefore with 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]

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 $\neq$ Z2).

[Please note the decomposition as given in the example, MVD is a formalised theory for the decomposition.]

3.8 MULTIVALUED NORMALISATION- FOURTH NORMAL FORM

We have considered an example of Programmer (Emp name, qualification, 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, qualifications) and P2(Emp name, languages) to overcome these problems. The decomposed relations are in fourth normal form (4NF), which we shall now define.

We are now ready to define 4NF. A relation R is in 4NF if, whenever a multivalued dependency **X ->> Y** holds then either

(a) the dependency is trivial, or

(b) X is a candidate key for R.

As noted earlier, the dependency **X ->> Φ? Or X ->> Y** in a relation R(X, Y) is trivial since they must hold for all R(X, Y). Thus, the decomposition proposed above is in 4NF.

In fourth normal form, we have a relation that has information about only one entity. If a relation has more than one multivalue attribute, we should decompose it to remove difficulties with multivalued facts.

Intuitively R is 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.

3.9 THE FIFTH NORMAL FORM

The basic principle behind 5NF is the situation when a table exists having all key attributes.

Consider the following relation SPJ (Supplier#, Part#, Project#).

The constraints are:

- A supplier can supply any part to any project.
- A project may need many parts supplied by any supplier.
- A part may be supplied by any supplier.

Let us consider the following instance of SPJ:

Relation: SPJ

Supplier#	Part#	proJect#
S1	P1	J1
S1	P2	J1
S1	P1	J3
S1	P2	J3
S2	P2	J3
S2	P2	J1
S2	P3	J1
S3	P1	J1
S3	P1	J2

In the above relation that Supplier S3 can supply Part2 cannot be represented till he supplies this part to some project, as the field proJect# is part of the only key to the relation which is (Supplier#, Part#, proJect#). On decomposing the relation into two tables SP and SJ we may encounter the following problem.

Decomposed First Relation: SP

Supplier#	Part#
S1	P1
S1	P2
S2	P2
S2	P3
S3	P1

Decomposed Second Relation: SJ

Supplier#	proJect#
S1	J1
S1	J3
S2	J3
S2	J1
S3	J1
S3	J2

On joining SP and SJ we get Relation: Lossy_SPJ

Supplier#	Part#	proJect#
S1	P1	J1
S1	P2	J1
S1	P1	J3
S1	P2	J3
S2	P2	J3
S2	P3	J3 ←
S2	P2	J1
S2	P3	J1
S3	P1	J1
S3	P1	J2

We need the third decomposed Relation: PJ

Part#	proJect#
P1	J1
P2	J1
P1	J3
P2	J3
P3	J1
P1	J2

On combining Lossy_SPJ with the PJ relation we will get the original relation.

There may be other problems relating to the 2 decomposibility of SPJ relation and it can be decomposed into three possible relations. This three decomposibility is best explained by **Join dependency** and **Fifth Normal Form**. However, in this course we will not present a formal definition of the two as they are the least properly understood. A designer have to rely more on his common sense to get this normal form.

3.10 RULES OF DATA NORMALIZATION

1. **Eliminate Repeating Groups:** Make a sep table for each set of related attributes, and give each table a primary key.
2. **Eliminate Redundant Data:** If an attribute depends on only part of a multi-valued key, remove it to a separate table.
3. **Eliminate Columns Not Dependent On Key:** If attributes do not contribute to a description of the key, remove them to a separate table.
4. **Isolate Independent Multiple Relationships:** No table may contain two or more 1:n or n:m relationships that are not directly related.
5. **Isolate Semantically Related Multiple Relationships:** There may be practical constrains on information that justify separating logically related many-to-many relationships.

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 member has experience with. Some might know many, and others might not know any.

Employee List				
Emp-ID	**Emp-Name**	**Dept-Name**	**Company-Loc**	**Database Known**
1	Gurpreet Malhotra	A	N-Delhi	Oracle
2	Faisal Khan	A	N-Delhi	Access
3	Manisha Kukreja	C	Agra	DBASE, Clipper, FoxPro
4	Sameer Singh	B	Bombay	DB2
5	Dave Jones	B	Bombay	DB2, Oracle

For the time being we are not considering Dept-Name and Company-Loc till step 3.

3.10.1 Eliminate Repeating Groups

If we give the query, "Find out list of employees ,who knows DB2".

For this query we need to perform an awkward scan of the list looking for references to DB2. This is inefficient and an extremely untidy way to retrieve information. Let us create another field termed as Databse-ID as D-ID to codify name of database in numeric form.

The Emp-ID in the database table matches the primary key in the employee table, providing a foreign key for relating the two tables with a join operation. Now we can answer the question by looking in the database table for "DB2" and getting the list of Employees.

Employee Table (Not shown Dept-Name and Company-Loc)	
Emp-ID	Emp-Name
1	Gurpreet Malhotra
2	Faisal Khan
3	Manisha Kukreja
4	Sameer Singh
5	Dave Jones

Database Table		
D-ID	Emp-ID	Database
1	2	Access
2	4	DB2
2	5	DB2
3	3	FoxPro
4	3	DBASE
5	3	Clipper
6	1	Oracle
6	5	Oracle

3.10.2 Eliminate Redundant Data

In the Database Table, the primary key is made up of the Emp-ID and the D-ID (Database ID). This makes sense for the "Where Learned" and "Skill Level" attributes, since they will be different for every member/database combination. But the database name depends only on the D-ID. The same database name will appear redundantly, every time its associated ID appears in the Database Table.

Let us now understand how does anomalies occur.

1. We need reclassify a database, thus we give different D-ID to the database. The change has to be made for every Employee that lists that database. If you miss some, you'll have several Employees with the same database under different IDs. This is an update anomaly.

2. Suppose the last Employee listing a particular database leaves the group. His records will be removed from the system, and the database will not be stored anywhere! This is a delete anomaly.

To avoid these problems, we need **second normal form**. To achieve this, we isolate the attributes depending on both parts of the key from those depending only on the D-ID. This results in two tables: "Database" which gives the name for each D-ID, and "EmpDatabase" which lists the databases for each member.

Employee Table (Not shown Dept-Name, Company-Loc)	
Emp-ID	Emp-Name
1	Gurpreet Malhotra
2	Faisal Khan
3	Manisha Kukreja
4	Sameer Singh
5	Dave Jones

EmpDatabase Table	
Emp-ID	D-ID
2	1
4	2
5	2
3	3
3	4
3	5
1	6
5	6

DataBase Table	
D-ID	Database
1	Access
2	DB2
3	FoxPro
4	DBASE
5	Clipper
6	Oracle

Now we can reclassify a database in a single operation: look up the D-ID in the "Database" table and change its name. The result will instantly be available throughout the application.

3.10.3 Eliminate Columns Not Dependent On Key

Now consider the fields Dept-Name and Company-Loc in the employee table.

The Employee Table satisfies

first normal form - As it contains no repeating groups.
second normal form - As it doesn't have a multivalued key.

But the key is Emp-ID; and the Dept-Name and location describe only about Department, not a Employee. To achieve third normal form, they must be moved into a separate table. Since they describe a department, Dept-Code becomes the key of the new "Company" table.

The motivation for this is the same for second normal form: we want to avoid update and delete anomalies. For example, suppose no Employees from the department "A" were currently stored in the database. With the previous design, there would be no record of its existence, even though 2 past members were from "A".

Employee-Table-Final	
Emp-ID	**Emp-Name**
1	Gurpreet Malhotra
2	Faisal Khan
3	Manisha Kukreja
4	Sameer Singh
5	Dave Jones

Department-Table		
Dept-ID	**Dept-Name**	**Company-Loc**
1	A	N-Delhi
2	B	Bombay
3	C	Agra
4	D	Banglore
5	E	Hyderabaad

Rest of the Table remains same.

3.10.4 Isolate Independent Multiple Relationships

Isolate Semantically Related Multiple Relationships

Example

Let us now simply consider an example, simply giving Normalisation Table.

Unnormalised Data Items for Puppies
Puppy number
Puppy name
Kennel Code
Kennel Name
Kennel location
Trick ID 1...n
Trick Name 1...n
Trick Where Learned 1...n
Skill Level 1...n

FIRST NORMAL FORM		
Puppy Table		
Puppy Number	Primary Key	Every puppy gets a unique number.
Puppy Name		
Kennel Name		
Kennel Location		
Trick Table		
Puppy Number	Primary Key [composite]	
Trick ID	Primary Key [composite]	
Trick Name		We'll add a row for every trick learned by every puppy.
Trick Where Learned		
Skill Level		

TRICK TABLE

Puppy Number	**Trick ID**	**Trick Name**	**Where Learned**	**Skill Level**
52	27	Roll Over	16	9
53	16	Nose Stand	9	9
54	27	Roll Over	9	5

SECOND NORMAL FORM		
Puppy Table		
Puppy Number	Primary Key	Every puppy gets a unique number.
Puppy Name		
Kennel Name		
Kennel Location		
Trick Table		
Trick ID		
Trick Name		We'll add a row for every trick learned by every puppy.
Puppy Tricks		
Puppy Number	Primary Key [composite]	
Trick ID	Primary Key [composite]	
Trick Where Learned		
Skill Level		

THIRD NORMAL FORM		
Puppy Table		
Puppy Number	Primary Key	Every puppy gets a unique number.
Puppy Name		
Kennel Code		
Kennels Table		
Kennel Code	Primary Key	
Kennel Name		
Kennel Location		
Trick Table		
Trick ID		
Trick Name		We'll add a row for every trick learned by every puppy.
Puppy Tricks		
Puppy Number	Primary Key [composite]	
Trick ID	Primary Key [composite]	
Trick Where Learned		
Skill Level		

Puppy Tricks and Costumes
Puppy Number
Trick ID
Trick Where Learned
Skill Level
Costume
FOURTH NORMAL FORM

Puppy Table		
Puppy Number	Primary Key	Every puppy gets a unique number.
Puppy Name		
Kennel Code		
Kennels Table		
Kennel Code	Primary Key	
Kennel Name		
Kennel Location		
Trick Table		
Trick ID		
Trick Name		We'll add a row for every trick learned by every puppy.

Puppy Tricks		
Puppy Number	Primary Key [composite]	
Trick ID	Primary Key [composite]	
Trick Where Learned		
Skill Level		
Costumes Table		
Costume Number	Primary Key	
Costume Name		
Puppy Costumes		
Puppy Number	Primary Key [composite]	
Costume Number	Primary Key [composite]	

Kennel-Breeder-Breeds (Not in 4NF)
Kennel Number
Breeder
Breed

Kennel-Breeder-Breeds (Not in 4NF)		
Kennel Number	**Breeder**	**Breed**
5	Acme	Spaniel
5	Acme	Dachshund
5	Acme	Banana-Biter
5	Puppy Factory	Spaniel
5	Puppy Factory	Dachshund
5	Puppy Factory	Banana-Biter
5	Whatapuppy	Spaniel
5	Whatapuppy	Dachshund
5	Whatapuppy	Banana-Biter

Kennel-Breed (4NF)	
Kennel Number	**Breed**
5	Spaniel
5	Dachshund
5	Banana-Biter

Kennel-Breeder (4NF)	
Kennel Number	**Breeder**
5	Acme
5	Puppy Factory
5	Whatapuppy

FIFTH NORMAL FORM		
Puppy Table		
Puppy Number	Primary Key	Every puppy gets a unique number.
Puppy Name		
Kennel Code		
Kennels Table		
Kennel Code	Primary Key	
Kennel Name		
Kennel Location		
Trick Table		
Trick ID	Primary Key	
Trick Name		We'll add a row for every trick learned by every puppy.
Puppy Tricks		
Puppy Number	Primary Key [composite]	
Trick ID		
Trick Where Learned		
Skill Level		
Costumes Table		
Costume Number	Primary Key	
Costume Name		
Puppy Costumes		
Puppy Number	Primary Key	
Costume Number	Primary Key	
Kennel-Breed		
Kennel Number	Primary Key	
Breed		
Kennel-Breeder		
Kennel Number	Primary Key	
Breeder		

Check Your Progress

Question 1. Let R be a relational of degree n. What is the maximum number of functional dependencies R can possibly satisfy (trivial as well as nontrivial)?

....

....

Question 2. List the set of all FDs satisfied (for all time) by shipments relation SPJ (defined in Question 1 of check your progress 2 in unit 1).

....

....

Question 3. A relation NADDR is defined with attributes NAME (unique), STREET, CITY, STATE and ZIP. For any given zipcode, there is just one city and state. Also, for any given street, city, and state, there is just one zipcode. Give an irreducible set of FDs for this relation. What are the candidate keys? What are the normalised tables, based on these FDs.

....

....

....

3.11 SUMMARY

A **Functional Dependency (FD)** is a many to-to-one relationship between two sets of attribute of a given relation. Given a relation R, the FD A → B (where A and B are subsets of the attributes of R) is said to hold in R if and only if, whenever two tuples of R have the same value for A, and they also have the same value for B.

The notion of functional dependency generalizes the notion of **superkey**.

Let R be a relation schema. A subset K of R is a superkey of R no two tuples in any legal relation r(R) may have the same value on attribute set K.

We discussed Single valued normalisation and discussed the concepts of **first**, **second**, **third**, and **Boyce/Codd normal forms**. The purpose of normalisation is to avoid **redundancy**, and hence to avoid certain **anamolies**.

We learned that the desirable properties of a decomposition of a base relation to it's normal forms, are that the decomposition should be attribute preserving, dependency preserving and lossless.

We also touched multivalued dependencies, although not in detail. We learned the various rules of data normalisation, which help to normalise the base relation very cleanly. Those rules are eliminating repeating groups, eliminate redundant data, eliminate columns not dependent on key, isolate independent multiple relationship and isolate semantically related multiple relationships.

3.12 MODEL ANSWERS

Check Your Progress

Question 1. An FD is basically a statement of the form A → B where A and B are each subsets of the set of attributes of R. Since a set of n elements has 2^n possible subsets, each of A and B has 2^n possible values, and hence an upper limit on the number of possible FDs is 2^{2n}.

Question 2. The complete set of FDs - for relation SPJ is as follows:
{ S#, P#, QUANTITY } → { S#, P#, QUANTITY }
{ S#, P#, QUANTITY } → { S#, P# }
{ S#, P#, QUANTITY } → { P#, QUANTITY }
{ S#, P#, QUANTITY } → { S#, QUANTITY }
{ S#, P#, QUANTITY } → { S#, }
{ S#, P#, QUANTITY } → { P }

{ S#, P#, QUANTITY } → { QUANTITY }
{ S#, P#, QUANTITY } → { }

{ S#, P# } → { S#, P#, QUANTITY }
{ S#, P# } → { S#, P# }
{ S#, P# } → { P#, QUANTITY }
{ S#, P# } → { S#, QUANTITY }
{ S#, P# } → { S#, }
{ S#, P# } → { P }
{ S#, P# } → { QUANTITY }
{ S#, P# } → { }

{ P#, QUANTITY } → { P#, QUANTITY }
{ P#, QUANTITY } → { P#, }
{ P#, QUANTITY } → { QUANTITY }
{ P#, QUANTITY } → { }

{ S#, QUANTITY } → { S#, QUANTITY }
{ S#, QUANTITY } → { S#, }
{ S#, QUANTITY } → { QUANTITY }
{ S#, QUANTITY } → { }

{ S# } → { S# }
{ S# } → { }

{ P# } → { P# }
{ P# } → { }

{ QUANTITY } → { QUANTITY }
{ QUANTITY } → { }

{ } → { }

Question 3. Abbreviating NAME, STREET, CITY, STATE, and ZIP as N, R, C, T, and Z, respectively, we have:

N → RCT
RCT → Z
Z → CT

An obviously equivalent irreducible set is:
N → R
N → C
N → T
RCT → Z
Z → C
Z → T

The only candidate key is N, as we can find any one of N, R, C, T, Z from N.

Let us first use the FD.

RCT → Z

We will get

R1 (Name, Street, City, State), R2 (Street, City, State, Zip) (3NF)

FDs N → R C T   RCT → Z, Z → C T

We may decompose R2 further using Z → C T

R21 (Street, Zip)   R22 (Zip, City, State)

but here the FD RCT → Z will be lost and is enforceable only on the join of the two relations. Although R21 and R22 will be in BCNF but it is not recommended and is a bad decomposition. Therefore, sometimes we may restrict ourselves to just 3NF.