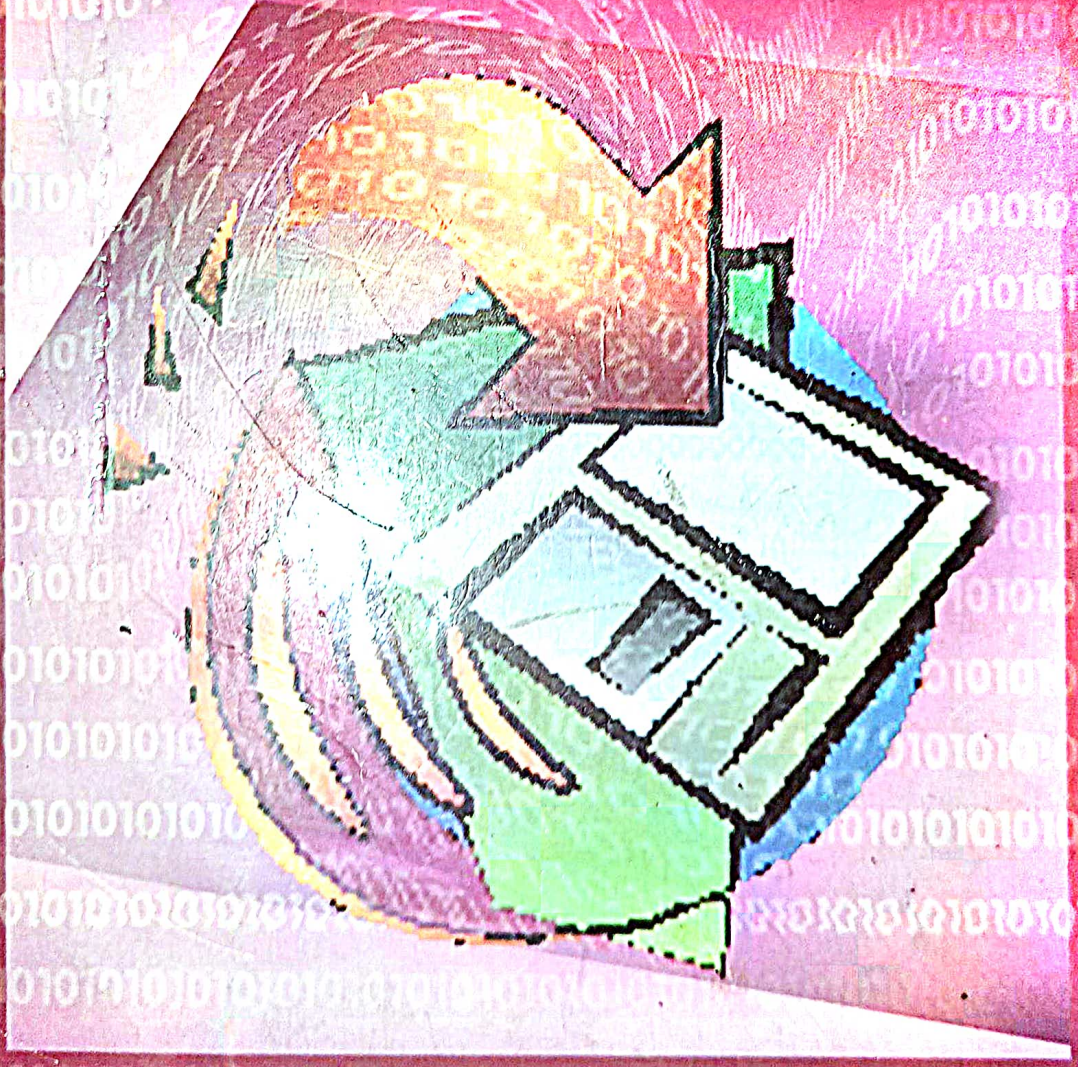
DATABASE MANAGEMENT SYSTEMS



Alexis Leon ♦ Mathews Leon

Relationships

There are three possible relationships between related entities or tables. They are:

- One-to-one
- One-to-many
- Many-to-many

Relational Databases are designed to work most effectively with one-to-many relationships between tables and expressed using primary and foreign keys. Defining a column as a primary key in the database will ensure that no duplicate values are contained in that column. In other words, a primary key will be unique.

One-to-one relationships are rare, because data elements related in this way are normally placed in the same table. When a one-to-one relationship is desired, it can be expressed simply as a one-to-many relationship using primary and foreign keys.

One-to-many relationships are the most common. A primary key is the 'one' side of the relationship, and the foreign key is the 'many' side of the relationship. A primary key value is unique in its own table. Each primary key may have any number of foreign keys using the same value, in any number of tables. In other words each pair of primary and foreign key columns is a one-to-many relationship. The data values stored in these columns determine which rows from the table containing the foreign key column are associated with an individual row in the table containing the primary key column. The foreign key value refers to one, and only one, value in the table containing the primary key. Multiple foreign keys may reference the primary key value. This is how one-to-many relationships are expressed and maintained by the database. The database can perform special operations on primary and foreign key relationship between tables, including the ability to delete all rows in a table containing a foreign key value when the associated primary key row is deleted or to restrict deletion of a row containing a primary key that has references in another table. Defining primary and foreign key columns in the database promotes data integrity.

Many-to-many relationships are problematic and cannot be adequately expressed directly in a relational database. Many-to-many relationships are usually expressed using intersection tables. An intersection table contains two or more foreign keys, relating the primary key values of two or more tables to each other. The role of an intersection table is to convert the many-to-many relationship into two one-to-many relationships that can be easily handled by the database.

FIRST NORMAL FORM (1NF)

First normal form is a relation in which the intersection of each row and column contains one and only one value. To transform the unnormalized table (a table that contains one or more repeating groups) to first normal form, we identify and remove the **repeating groups** within the table. A repeating group is a set of columns that store similar information that repeats in the same table. Consider the following table, which contains the contact tracking information:

CREATE TABLE CONTACTS

```
(CONTACT_ID          INTEGER          NOT NULL,
L_NAME               VARCHAR(20)      NOT NULL,
F_NAME               VARCHAR(20)
CONTACT_DATE1        DATE,
CONTACT_DESC1        VARCHAR(50),
CONTACT_DATE2        DATE,
CONTACT_DESC2        VARCHAR(50));
```

The above data structure contains a repeating group of the date and description of two conversations. The only advantage of designing the table like this is that it avoids the need for a relationship. But the disadvantages are many:

- This structure limits the number of conversations to two, which will create problems when more than two conversations need to be stored.
- This structure also makes it difficult to do any kind of meaningful searching using the columns, for example to locate a conversation on a specific date (here both the date columns have to be searched which will result is a clumsy SQL code).

To eliminate the repeating group, the group is moved to another table, which is then related to the parent table. The primary key of the parent table (CONTACT_ID) is stored in the second table. Moving the repeating group into another table allows any number of conversations to be recorded and searched easily. The primary and foreign key relationships are defined to ensure that conversations that do not relate to a contact are not recorded. The new arrangement is shown in Figure 11.1 and the DDL statements for creating the tables are given after that.

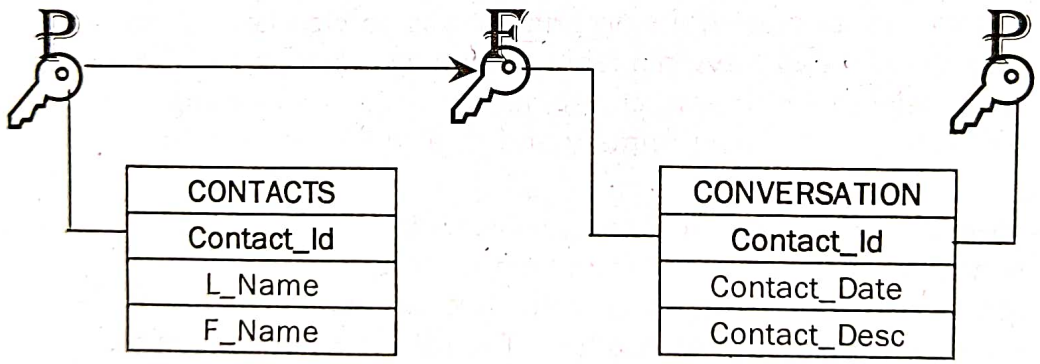


Figure 11.1 **CONTACTS-CONVERSATION Relationship**

```
CREATE TABLE CONTACTS
(CONTACT_ID          INTEGER          NOT NULL,
L_NAME               VARCHAR(20)      NOT NULL,
F_NAME               VARCHAR(20));

CREATE TABLE CONVERSATION
(CONTACT_ID          INTEGER          NOT NULL,
CONTACT_DATE         DATE,
CONTACT_DESC         VARCHAR(50));

ALTER TABLE CONTACTS
```

```
ADD CONSTRAINT CONTACT_ID_PK
PRIMARY KEY (CONTACT_ID);

ALTER TABLE CONVERSATION
ADD CONSTRAINT CONTACT_ID_FK
FOREIGN KEY (CONTACT_ID) REFERENCES CONTACTS (CONTACT_ID);
```

In summary, every table should have a primary key, and each set of repeating groups should appear in its own table. When these criteria are satisfied, we say that the first normal form is achieved.

SECOND NORMAL FORM (2NF)

A relation is in second normal form if it is in first normal form and every non-key attribute is fully and functionally dependent on the primary key. Thus no non-key attribute is functionally dependent on the primary key. A relation in the first normal form will be in the second normal form if one of the following conditions is satisfied:

- The primary key consists of only one attribute (field or column).
- No non-key attributes exist in the relation. In other words, all the attributes in the relation are components of the primary key.
- Every non-key attribute is functionally dependent on the full set of primary key attributes.

Consider a relation called EMPLOYEE with the attributes Emp_ID, Department, F_Name, L_Name, Salary and Date-of-birth. Here assume that the primary key for this relation is the composite key Emp_ID + Department. In this case the non-key attributes such as F_Name, L_Name, Salary and Date-of-birth are functionally dependent on part of the primary key (Emp_ID) but not on the Department. A partial functional dependency is a functional dependency in which one or more non-key attributes (such as F_Name, L_Name, Salary or Date-of-birth) are functionally dependent on part (but not all) of the primary key. The partial functional dependency in the EMPLOYEE table creates a redundancy in that relation, which results in anomalies when the table is updated.

Redundant data is data that is unnecessarily expressed multiple times or that depends only on part of a multi-value key. In other words, when a column's value is dependent on the value of one column in the table, but not another, it is considered redundant. For example:

```
CREATE TABLE EMPLOYEE
(EMP_NO          INTEGER          NOT NULL,
L_NAME           VARCHAR(20)      NOT NULL,
F_NAME           VARCHAR(20),
DEPT_CODE        INTEGER,
DESCRIPTION      VARCHAR(50));
```

This table contains redundant data, namely the department description, which depends only on the DEPT_CODE and does not vary based on the value of the EMP_NO, which is the

primary key of the table. So by storing the department code and description in a different table the redundancy is elliminated.

```
CREATE TABLE EMPLOYEE
(EMP_NO          INTEGER          NOT NULL,
L_NAME           VARCHAR(20)      NOT NULL,
F_NAME           VARCHAR(20),
DEPT_CODE        INTEGER);

CREATE TABLE DEPARTMENT
(DEPT_CODE       INTEGER          NOT NULL,
DESCRIPTION      VARCHAR(50)      NOT NULL);
```

This relationship is shown in Figure 11.2.

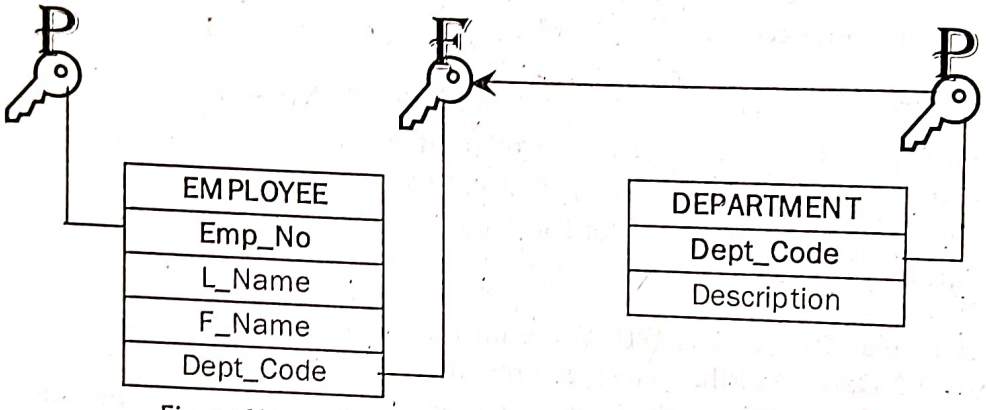


Figure 11.2 **EMPLOYEE-DEPARTMENT Relationship**

THIRD NORMAL FORM (3NF)

A relation is in third normal form if it is in second normal form and no transitive dependencies exist. A transitive dependency in a relation is a functional dependency between two or more non-key attributes. The columns in each table should be a group of columns in which the data in each column contributes to the description of each row in the table. For a given row with a unique key, each column appearing in that row should contribute to the description of that row. For example in the following table, the columns F_NAME and L_NAME contribute to describing a specific contact using the primary key CONTACT_ID. But, the COMPANY_NAME and COMPANY_LOCATION do not contribute to describing the record with a given CONTACT_ID, since it identifies an individual and not a company.

```
CREATE TABLE CONTACTS
(CONTACT_ID      INTEGER          NOT NULL,
L_NAME           VARCHAR(20)      NOT NULL,
F_NAME           VARCHAR(20),
COMPANY_NAME     VARCHAR(20),
```

```
COMPANY_LOCATION    VARCHAR(50));
```

In the above relation CONTACT_ID is the primary key, so that all the remaining attributes are functionally dependent on this attribute. However there is a transitive dependency—COMPANY_LOCATION is dependent on COMPANY_NAME and COMPANY_NAME is functionally dependent on CONTACT_ID. So unless the location of the company differs on an individual basis, this column in not dependent on the key value and should be removed to another table. Here one thing that should be noted is that, as a result of the transitive dependency, there are update anomalies in the CONTACTS table as follows:

- **Insertion anomaly** – A new company cannot be inserted to the CONTACTS table until a contact person has been assigned to that company.
- **Deletion anomaly** – If a company that has only one contact person is deleted from the table, we will lose the information about that company, as the company information is associated with that person.
- **Modification anomaly** – If a company changes its location we will have to make the change in all the records where the company name appears. Suppose, if the company has five contact persons, then we will have to make the changes in five places.

The above anomalies arise as a result of the transitive dependency. The transitive dependency can be removed by decomposing the above table into two as shown in Figure 11.3.

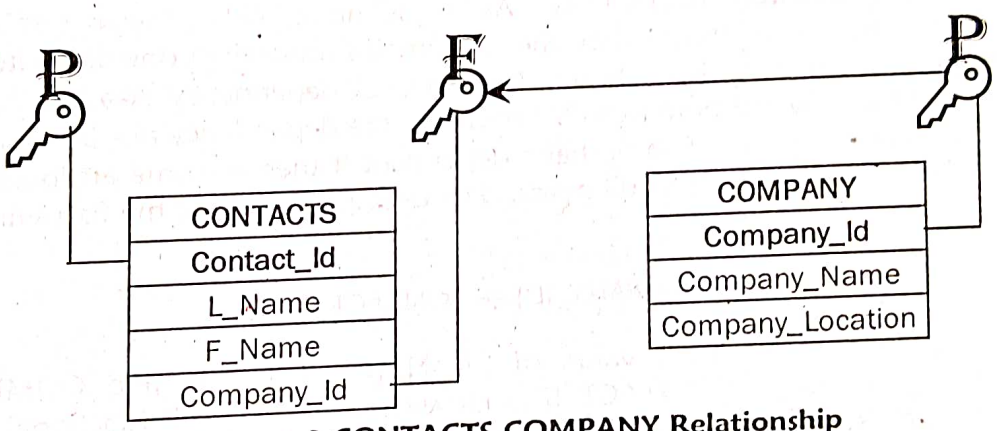


Figure 11.3 **CONTACTS-COMPANY Relationship**

```
CREATE TABLE CONTACTS
(CONTACT_ID          INTEGER          NOT NULL,
L_NAME               VARCHAR(20)      NOT NULL,
F_NAME               VARCHAR(20),
COMPANY_ID           INTEGER);

CREATE TABLE COMPANY
(COMPANY_ID          INTEGER          NOT NULL,
COMPANY_NAME         VARCHAR(20)      NOT NULL,
COMPANY_LOCATION     VARCHAR(50));
```

When all the columns in a table describe and depend upon the primary key, the table is said to satisfy the third normal form.

BOYCE-CODD NORMAL FORM (BCNF)

Database relations are designed so that they have neither partial dependencies nor transitive dependencies, because these types of dependencies result in update anomalies. A functional dependency describes the relationship between attributes in a relation. For example, if 'A' and 'B' are attributes in relation R, 'B' is functionally dependent on 'A' (denoted by A→B), if each value of 'A' is associated with exactly one value of 'B'. For example in the CONTACTS table, we can say that L_NAME, F_NAME and COMPANY_ID are functionally dependent on CONTACT_ID. These dependencies are expressed as follows:

- CONTACT_ID → L_NAME
- CONTACT_ID → F_NAME
- CONTACT_ID → COMPANY_ID
- CONTACT_ID → {L_NAME, F_NAME, COMPANY_ID}
- {L_NAME, F_NAME, COMPANY_ID} → CONTACT_ID

The left-hand side and the right-hand side of a functional dependency are sometimes called the **determinant** and **dependent** respectively. As the definition states, the determinant and the dependent are both, sets of attributes. When the set contains more than one attribute we will use the braces to enclose them as shown above. A functional dependency A→B is full functional dependency if removal of any attribute from 'A' results in the dependency not being sustained any more. A functional dependency A→B is partially dependent if there is some attribute that can be removed from 'A' and the dependency still holds. For example, consider the following functional dependency:

{L_NAME, F_NAME, COMPANY_ID} → CONTACT_ID

It is correct to say that each value of {L_NAME, F_NAME and COMPANY_ID} is associated with a single value of CONTACT_ID. However, it is not full functional dependency because CONTACT_ID is also functionally dependent on a subset of {LNAME, F_NAME and COMPANY_ID}, namely {L_NAME and F_NAME}.

Transitive dependency, as we have seen, is a condition where A, B and C are attributes of a relation such that if A→B and B→C, then C is transitively dependent on A via B (provided that A is not functionally dependent on B or C). Transitive dependency is a description of a type of functional dependency that occurs when the following functional dependencies hold between A, B and C of a relation:

A→B and B→C

Then the transitive dependency A→C exists via attribute B. This condition holds provided that A is not functionally dependent on B or C. For example, consider the following functional dependencies within an EMPLOYEE-DEPARTMENT relationship. The EMPLOYEE relation has

attributes like EMP_NO, NAME, ADDRESS, POSITION, SALARY, DEPT_ID, etc. The DEPARTMNET relation has attributes like DEPT_ID, DEPT_NAME, MANAGER and so on. Now consider the following dependencies:

EMP_NO→DEPT_ID and DEPT_ID→DEPT_NAME

Then the transitive dependency EMP_NO→DEPT_NAME exists via DEPT_ID attribute. This condition holds, as EMP_NO is not functionally dependent on DEPT_ID or DEPT_NAME.

One of the major aims of relational database design is to group attributes into relations so as to minimize data redundancy and thereby reduce the file storage space required by the implemented base relations. Relations that have redundant data may have problems called **update anomalies**, which are classified as insertion, deletion or modification anomalies. These anomalies occur because, when the data in one table is deleted or updated or new data is inserted, the related data is also not correspondingly updated or deleted. Sometimes when a deletion in one table occurs, it will leave meaningless data in other tables. One of the aims of the normalization is to remove the update anomalies.

Boyce-Codd Normal Form (BCNF) is based on functional dependencies that take into account all candidate keys in a relation. For a relation (table) with only one candidate key, third normal form and BCNF are equivalent. **A relation is in BCNF if and only if every determinant is a candidate key.** To test whether a relation is in BCNF, we identify all the determinants and make sure that they are candidate keys. A determinant is an attribute or a group of attributes on which some other attribute is fully functionally dependent.

The difference between third normal form and BCNF is that for a functional dependency A→B, the third normal form allows this dependency in a relation if 'B' is a primary-key attribute and 'A' is not a candidate key, whereas BCNF insists that for this dependency to remain in a relation, 'A' must be a candidate key. Therefore BCNF is a stronger form of the third normal form, such that every relation in BCNF is also in the third normal form. However, a relation in the third normal form is not necessarily in BCNF.

Consider the following relation INTERVIEW, which contains the details of the arrangements for interviews of candidates by the in-house technical experts of a company. The interviewers are allocated a specific room on the day of the interview. A room can be allocated to several interviewers as required, throughout the day. A candidate is interviewed only once on a given date. The relation is shown below:

Table 11.1 **INTERVIEW Table**

INTERVIEW				
CANDIDATE_ID	**INT_DATE**	**INT_TIME**	**INTVR_ID**	**ROOM_NO**
C001	24-May-1999	10.30	E001	R001
C002	24-May-1999	11.30	E001	R001
C003	24-May-1999	10.30	E002	R002
C004	26-May-1999	11.30	E003	R002

The above table has three composite candidate keys, which overlap by sharing the common attribute INT_DATE as shown below:

1. CANDIDATE_ID and INT_DATE

2. INTVR_ID, INT_DATE and INT_TIME
3. ROOM_NO, INT_DATE and INT_TIME

We make the composite candidate key (CANDIDATE_ID, INT_DATE) as the primary key. Now the relation has the following functional dependencies:

- {CANDIDATE_ID, INT_DATE} →{INT_TIME, INTVR_ID, ROOM_NO} — Primary Key
- {INTVR_ID, INT_DATE, INT_TIME} →CANDIDATE_ID — Candidate Key
- {ROOM_NO, INT_DATE, INT_TIME} →{INTVR_ID, CANDIDATE_ID} — Candidate Key
- {INTVR_ID, INT_DATE} →ROOM_NO

The first three dependencies are all candidate keys for this relation and will not cause any problems for the relation. The only functional dependency that requires discussion is the fourth one—{INTVR_ID, INT_DATE}→ROOM_NO. Even though the {INTVR_ID, INT_DATE} is not a candidate key for the relation, this functional dependency is allowed in the third normal form, because ROOM_NO is a primary key attribute being part of the candidate key— ROOM_NO, INT_DATE and INT_TIME. As there are no partial or transitive dependencies on the primary key (CANDIDATE_ID and INT_DATE) and since the functional dependency {INTVR_ID, INT_DATE}→ROOM_NO is allowed, the relation is in the third normal form.

However, this relation is not in BCNF (a stronger form of the third normal form), due to the presence of the {INTVR_ID, INT_DATE} determinant, which is not a candidate key for the relation. BCNF requires that all the determinants in a relation must be a candidate key for the relation. As a consequence the INTERVIEW relation may suffer from update anomalies. For example, to change the room number for Interviewer E001 on 24-May-1999, we need to update two rows. If only one row is updated with new room number, this results in an inconsistent state for the database. Even though it is perfectly possible that the same interviewer can conduct the interview in different rooms it is against our original assumption that an interviewer is assigned a room for the day. So to transform the INTERVIEW relation to BCNF, we must remove the violating functional dependency by creating two new relations INTERVIEW and ROOM as shown below:

Table 11.2 **INTERVIEW Table**

INTERVIEW			
CANDIDATE_ID	**INT_DATE**	**INT_TIME**	**INTVR_ID**
C001	24-May-1999	10.30	E001
C002	24-May-1999	11.30	E001
C003	24-May-1999	10.30	E002
C004	26-May-1999	11.30	E003

Table 11.3 **ROOM Table**

ROOM		
INTVR_ID	**INT_DATE**	**ROOM_NO**
E001	24-May-1999	R001
E002	24-May-1999	R002
E003	26-May-1999	R002

FOURTH NORMAL FORM (4NF)

A group of tables that satisfies the first, second and third normal forms are sufficiently well designed. However, isolating independent multiple relationships will further improve the data model when one-to-many and many-to-many relationships between tables are involved. In other words, no table should contain two or more one-to-many or many-to-many relationships that are not directly related to the key. These kinds of relationships are called **multi-valued dependencies** (MVDs). Multi-valued dependencies are the result of the first normal form, which prohibited an attribute from having a set of values. If we have two or more multi-valued independent attributes in the same relation (table), we get into a situation where we have to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain independence among the attributes involved. This constraint is specified by a multi-valued dependency.

Consider a table (relation) EMPLOYEE that has the attributes Name, Project and Hobby. A row in the EMPLOYEE table represents the fact that an employee works for a project and has a hobby. But an employee can work in more than one project and can have more than one hobby. The employee's projects and hobbies are independent of one another. To keep the relation state consistent we must have a separate tuple to represent every combination of an employee's project and an employee's hobbies. This constraint is specified as a multi-valued dependency on the EMPLOYEE relation. So whenever two independent one-to-many relationships (A:B and A:C) are mixed in the same relation, a multi-valued dependency arises. We will see the employee table and how the multi-valued dependency can be avoided using the fourth normal form. Given below is the EMPLOYEE table and its contents:

Table 11.4 **EMPLOYEE Table**

EMPLOYEE		
NAME	**PROJECT**	**HOBBY**
Alexis	Microsoft	Reading
Alexis	Oracle	Music
Alexis	Microsoft	Music
Alexis	Oracle	Reading
Mathews	Intel	Movies
Mathews	Sybase	Riding
Mathews	Intel	Riding
Mathews	Sybase	Movies

As we have seen, the above relation has two multi-valued dependencies—(Name, Project) and (Name, Hobby). We resolve this by decomposing the EMPLOYEE table into two tables that satisfy the fourth normal form as follows:

Table 11.5 **PROJECT and HOBBY Tables**

PROJECT	
NAME	**PROJECT**
Alexis	Microsoft
Alexis	Oracle
Mathews	Intel
Mathews	Sybase

HOBBY	
NAME	**HOBBY**
Alexis	Reading
Alexis	Music
Mathews	Movies
Mathews	Riding

Now consider another table—CONTACTS:

```
CREATE TABLE CONTACTS
(CONTACT_ID          INTEGER          NOT NULL,
L_NAME               VARCHAR(20)      NOT NULL,
F_NAME               VARCHAR(20));

CREATE TABLE COMPANY
(CONTACT_ID          INTEGER          NOT NULL,
COMPANY_NAME         VARCHAR(20),
COMPANY_LOCATION     VARCHAR(50)
PHONE_NUMBER         VARCHAR(20));
```

Even though this is not a violation of the third normal form this data structure makes it difficult to store a phone number of a contact without a company and makes it impossible to store multiple phone numbers for the same contact without repeating the company information.

The Fourth Normal Form is violated because there are two independent relationships represented in the COMPANY table—one between CONTACTS and COMPANIES and another between CONTACT and PHONE_NUMBER, which should be avoided. This can be achieved as shown in Figure 11.4 and defined in the DDL that follows.

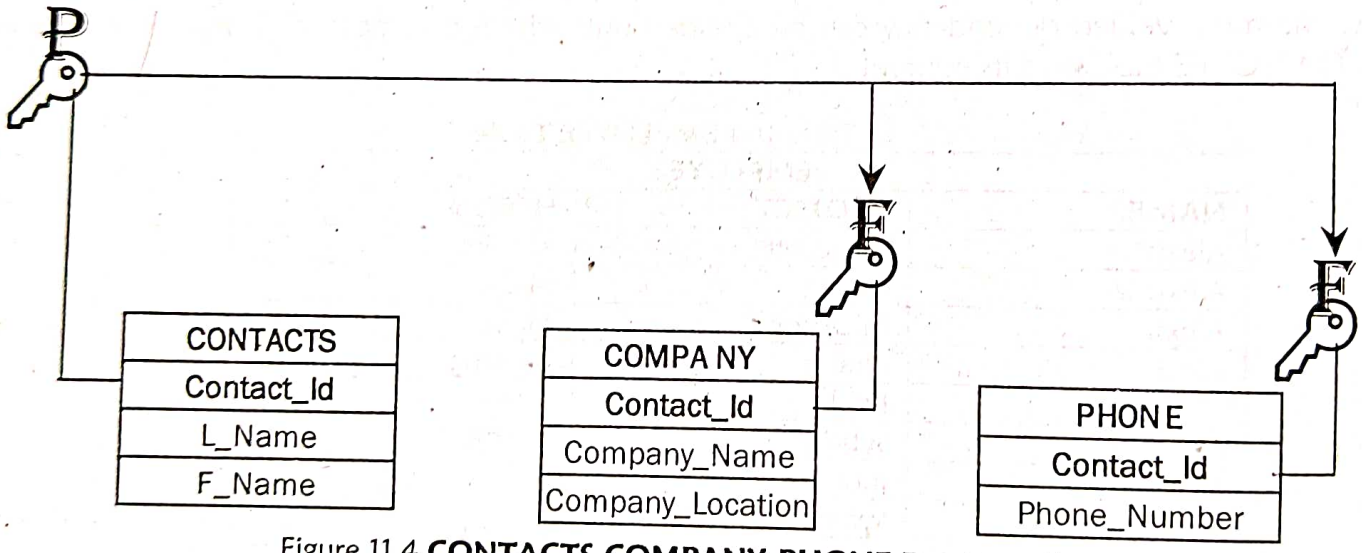


Figure 11.4 **CONTACTS-COMPANY-PHONE Relationship**

```
CREATE TABLE CONTACTS
(CONTACT_ID          INTEGER          NOT NULL,
L_NAME               VARCHAR2(20)     NOT NULL,
F_NAME               VARCHAR2(20));

CREATE TABLE COMPANY
(CONTACT_ID          INTEGER          NOT NULL,
COMPANY_NAME         VARCHAR2(20),
COMPANY_LOCATION     VARCHAR2(50)
```

```
CREATE TABLE PHONE
(CONTACT_ID          INTEGER              NOT NULL,
PHONE_NUMBER         VARCHAR2(20));
```

This allows companies and phone numbers to have independent, one-to-many relationships with CONTACTS table, which satisfies the fourth normal form.

FIFTH NORMAL FORM (5NF)

The chances that you will ever get to use the fifth normal form are very few, because it requires semantically related multiple relationships, which are rare. Semantically related multiple relationships are two or more relationships among tables that are related closely enough so that they can be resolved into a single relationship. Fifth normal form specifies that they remain separate.

Consider the chemical analysis labs, which test products of various companies. Three tables exist, LAB, PRODUCT and COMPANY. Companies can offer one or more products, labs can test one or more products from one or more companies and a product can be tested in one or more labs and can be offered by one or more companies. Whenever a lab is equipped to test a product from any company, the lab is equipped to test the same product from all companies. If a lab is equipped to analyze a product from a given company, this can be recorded in a table with the following structure:

```
CREATE TABLE LAB_PRODUCT_COMPANY
(LAB_ID          INTEGER      NOT NULL,
PRODUCT_ID       INTEGER      NOT NULL,
COMPANY_ID       INTEGER      NOT NULL);
```

The table contains three foreign keys expressing two relationships: the relationship between LABS and PRODUCTS and that between LABS and COMPANIES. The relationships are semantically related because they can be expressed using the same table. In other words, the relationship between PRODUCTS and COMPANIES is implied and expressed in this table.

There is nothing wrong with the above data structure, but it does not satisfy the fifth normal form and more than necessary entries are required in the table if the relationships are separated. Because a lab can test the same product for all companies who offer a given product, the following structure will be better:

```
CREATE TABLE LAB_PRODUCT
(LAB_ID          INTEGER      NOT NULL,
PRODUCT_ID       INTEGER      NOT NULL);

CREATE TABLE LAB_ COMPANY
(LAB_ID          INTEGER      NOT NULL,
COMPANY_ID       INTEGER      NOT NULL);
```

This allows the products that a given lab can test to be recorded and the companies whose products can be tested by a lab to be recorded with fewer entries. For example if lab A is newly equipped to do a spectroscopic analysis of products A, B and C offered by companies X and Y, the new structure requires five entries, 3 in the LAB_PRODUCT table and two in LAB_COMPANY table, to express this. To achieve the same, with the original structure we will require six entries.

The fifth normal form is not frequently invoked for the reason that the situation simply does not arise frequently.