CHAPTER 1

An Overview of DBMS and DB Systems Architechture

1.1 Introduction to Database Management Systems

A *Database Management System* (DBMS) is the software system that allows users to define, create and maintain a database and provides controlled access to the data. A database is a logically coherent collection of data with some inherent meaning. The term *database* is often used to refer to the data itself; however, there are other additional components that also form part of a complete database management system. Figure 1-1 shows that a complete DBMS usually consists of hardware, software including utilities, data, users and procedures. These items will be explained in the following paragraphs.

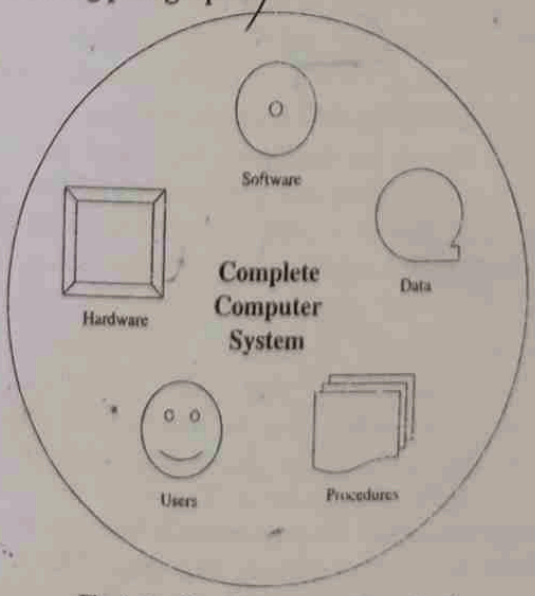


Fig. 1-1. Complete computer system.

The *hardware* is the actual computer system used for keeping and accessing the database. In large organizations, the hardware for such a system typically consists of a network with a central server and many client programs running on desktops. The server is the central processor where the database is physically located. The server usually has a more powerful processor because it handles the data retrieval operations and most of the actual data manipulation. The clients are the programs that interact with the RDBM and run on the personal desktops at the user's end to access the database. A DBMS and its clients can also reside in a single computer. In that case there is usually only one user at a time accessing the database, either a single user or a single personal database management system accessed by several users at different times. The actual configuration of the network varies from organization to organization. Specific hardware issues will not be addressed in this book.

The *software* is the actual DBMS. In a client/server network, the DBMS allows for data handling programs residing on the server and client programs on each desktop. In a single-user system usually only one piece of software handles everything. The DBMS allows the users to communicate with the database. In a sense, it is the mediator between the database and the users. Each client station or each individual user can be given different levels of access to the data. Some will be allowed to change portions of the database structure, some can change the existing data, and others will only be allowed to view the data. The DBMS controls access and helps maintain the consistency of the data. Utilities are usually included as part of the DBMS. Some of the most common utilities are report writers, application development tools and other design aids. Examples of DBMS software include Microsoft Access™, Oracle Corporation Personal Oracle™, and IBM DB2™. The typical arrangement of all the software modules and how they interact in a DBMS will be explained at the end of this chapter.

The database should contain all the *data* needed by the organization. One of the major features of databases is that the actual data are separated from the programs that use the data. The set of facts represented in a database is called the *Universe of Discourse* (UOD). The UOD should only include facts that form a logically coherent collection and that are relevant to its users. For this reason, a database should always be designed, built and populated for a particular audience and for a specific purpose. Probably as part of the UOD discussion, it is important to point out that only a partial view of the real world can be captured by a DBMS. Emphasis is on the relevant data pertaining to one or more objects, or *entities*. We will define an entity as the thing of significance about which information needs to be known. The characteristics that describe or qualify an entity are called *attributes* of the entity. For instance, in a student database, the basic entity is the student. Information recorded about that entity might be first and last name, major, grade point average, home address, current address, date of birth, and class level. These are the attributes of the student entity. The system would not be interested in the type of clothes, the number of friends, the movies the student attends, and so on. That is, this information is not relevant to the user and should not be part of the UOD. More explanation concerning data will be given in the next section.

CHAPTER 1 An Overview of DBMS and DB

Also, for each attribute, the set of possible values that the attribute can take is called the *domain* of the attribute. The domain of the date of birth would be all the dates that might be reasonable in the student body; none in the 1700s would be expected. Undergraduate class levels would probably be restricted to Freshman, Sophomore, Junior, and Senior. No other values would be allowed for that attribute.

There are a number of *users* who can access or retrieve data on demand using the applications and interfaces provided by the DBMS. Each type of user needs different software capabilities.

- The *database administrator* (DBA) is the person or group in charge of implementing the database system within the organization. The DBA has all the system privileges allowed by the DBMS and can assign (grant) and remove (revoke) levels of access (privileges) to and from other users.
- The *end users* are the people who sit at workstations and interact directly with the system. They may need to respond to requests from people outside the organization, to find answers quickly to questions from higher-level management, or to generate periodic reports. In some cases the end users should be allowed to change data within the system, for example addresses or order information. Other end users, such as those at a help desk, would only need privileges to view the data, not to change it.
- The *application programmers* interact with the database in a different manner. They access the data from programs written in high-level languages such as Visual Basic or C++. The application programmers design systems such as payroll, inventory, and billing that normally need to access and change the data.

An integral part of any system is the set of *procedures* that control the behavior of the system, that is, the actual practices the users follow to obtain, enter, maintain, and retrieve the data. For example, in a payroll system, how are the hours worked received by the clerk and entered into the system? Exactly when are monthly reports generated and to whom are they sent? These procedures are often formalized so that users at any level know exactly what to do and how to do the assigned task. In many organizations, if some employees have been there for a long time they may know exactly what to do and when. However, it is important to have procedures clearly articulated and written on record so that the system would not be jeopardized if new employees needed to use the system. Part of the job of the DBA is to verify that all procedures related to the complete system are clearly delineated.

Example 1.1

Indicate which type of user would perform the following functions for a payroll system in a large company: (a) Write an application program to generate and print the checks, (b) change the address in the database for an employee who has moved, (c) create a new user account for a newly hired payroll clerk.

a. Write an application program to generate and print the checks.

An application programmer or a team of programmers would design and implement such an application program.

b. Change the address in the database for an employee who has moved.

An end user might take the information from the employee over the phone and directly access the database to change it. However, changing such information from phone conversations may result in incorrect data because of typographical error or misunderstanding. In order to verify that updates are made correctly, the procedures in many organizations require that database changes be submitted in writing.

c. Create a new user ID for the newly hired payroll clerk.

The DBA or the DBA assistant working under the supervision of the DBA would be the person to create new user IDs. In a small organization, there might be only one person who does all administration of the system. In larger organizations, DBA assistants on the database administration team would be assigned different jobs. One person might handle all user accounts and another might be in charge of database maintenance.

1.1.1 DATA

The data are the heart of the DBMS. There are two kinds of data. First, and most obvious, is the collection of information needed by the organization. The second kind of data, or *metadata*, is information about the database. This information is usually kept in a *data dictionary* or *catalog*. The data dictionary includes information about users, privileges and the internal structure of the database. Careful management of all the data is essential in order that information can be trusted to be up to date and accurate. All levels of users need to have a firm understanding of the database and how it is structured. It is helpful to examine the database from several different perspectives. The system may be multi-user or single-user; the data are usually both integrated and shared; and the database may be centralized or distributed.

First, the configuration of the hardware and the size of the organization will determine whether it is a *multi-user system* or a *single-user system*. In a single-user system, the database resides on one computer and is only accessed by one user at a time. This one user may design, maintain, and write programs for the system, performing all the user roles. On the other hand, someone else, often a consultant, may have been hired to design the system. In this case, the single user may only perform the role of end user and the data may always be accessed interactively through the DBMS without using application programs.

Because of the large amount of data managed even by small organizations, most systems are multi-user. In this situation, the data are both *integrated* and

shared. A database is integrated when the same information is not recorded in two places. For example, both the billing department and the shipping department may need customer addresses. Even though both departments may access different portions of the database, the customers' addresses should only reside in one place. It is the job of the DBA to make sure that the DBMS makes the correct addresses available from one central storage area.

Likewise, the individual pieces of data are shared by both departments. The DBMS must assure that the two users are not changing different portions of the data at the same time. If this happens, the data may not remain accurate. Also, users who share data do not need the same level of access. The shipping department may only need to examine the customer's address for shipping purposes and should have no need to examine the customer's payment history. The billing department needs to be able both to examine the current balance and to change the balance when a payment is made. These permissions are called *privileges* and, as indicated before, are assigned by the DBA.

Example 1.2
Consider a database at a cable company which contains customer names, addresses, service categories (basic cable, premium channels, pay-per-view, etc.) and billing information. Indicate for each user, a billing clerk, a repair person, and a customer service representative, which items that user should be able to access and which items the user should be able to access and change.

User	Permission Level
a. Billing clerk	Should be able to access and change all data.
b. Repair person	Needs to access but not change name, address, and service information. Should not have access to any billing information.
c. Customer service representative	Needs to be able to access and change name, address, and service information. If billing questions are referred to the billing department, the customer service person has no need of any billing information.

A third issue for understanding both the data and the DBMS is whether the system is *centralized* or *distributed*. During the 1970s and 1980s most database management systems resided on large mainframes or minicomputers. The systems were centralized and *single tier*, which means the DBMS and the data reside in one location. The theory was that if data are kept in two places there is a high probability that two items which are supposed to be identical may not in fact be the same. For example, if a customer's address is stored in two separate tables for any reason, it is possible for one to be changed and the other to remain the same. Often dumb terminals were used to access the DBMS through teleprocessing.

The use of personal computers in businesses during the 1980s, the increased reliability of networking hardware, and the advances of doing business over the Internet during the 1990s led to the newer trend of trying to maintain accuracy of data and still make use of distributed systems. Two-tier and three-tier systems became common. In a *two-tier* system, different software is required for the server and for the client. The *three-tier* system adds middleware, which provides a way for clients of one DBMS to access data from another DBMS. Fig. 1-2 illustrates the difference between single-tier, two-tier and three-tier software systems.

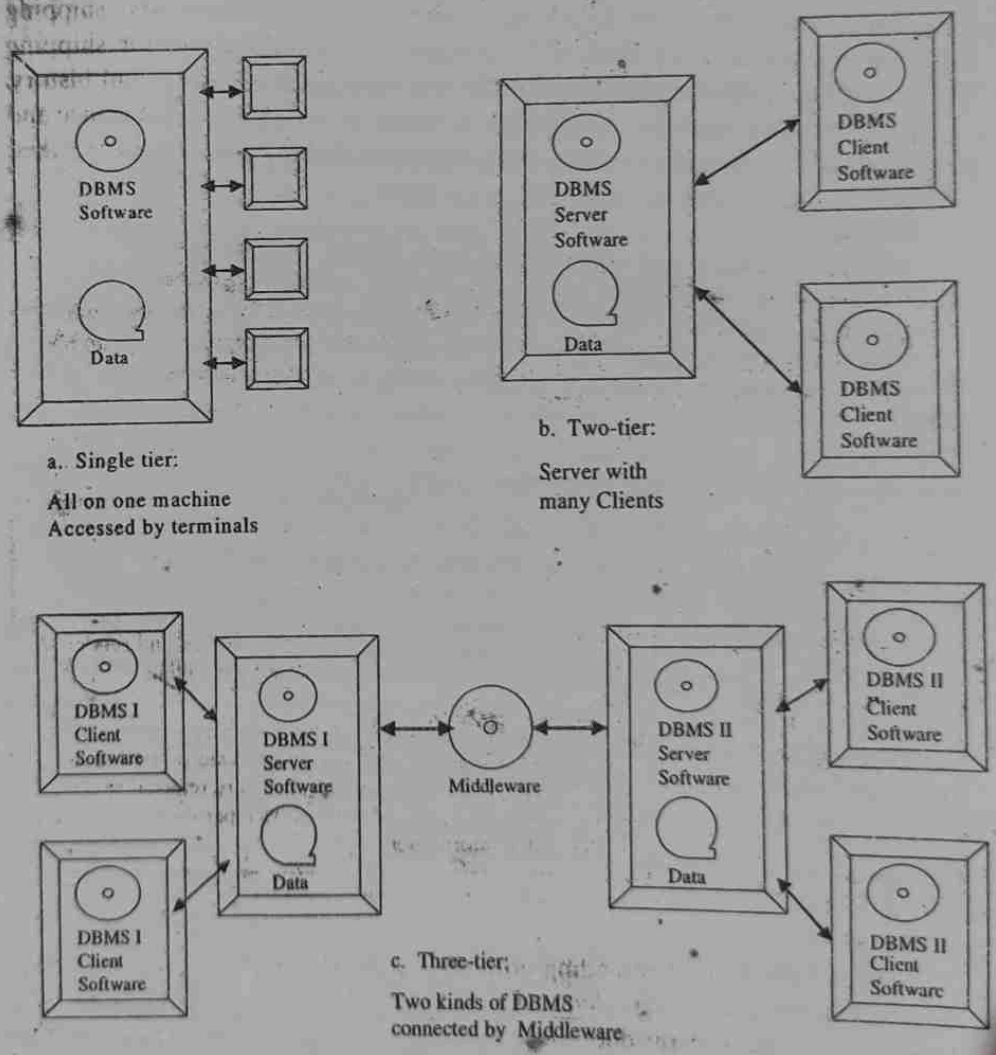


Fig. 1-2. Single-, two- and three-tier configurations.

A distributed DBMS can be implemented in several different ways. A local office network may store the customer data on one server and the supplier data on another. In both cases, the system would allow many client programs to access the data from both servers at the same time. Chapter 6 discusses the security

implications of this configuration. Another method of distribution is to store several equivalent databases in different places. The data are distributed geographically and located closest to where they will be used. However, it is critical in a distributed database that each node of the distributed system should be able to execute a global application or access files at any other node. For example, an organization with branches in several states may store a different customer list at each branch. The tables are distributed but connected, so the DBMS is able to find the information for any customer at any time from any location. The users ask for particular information and the DBMS hides the details of how it locates the requested data. This transparency is an important issue in distributed DBMS software. Remember, even though the database may be distributed, it is not the same as being decentralized. Items of data still reside in only one place and the DBMS knows where to find them. Another advantage of the distributed model is that it results in improved reliability and performance. When both data and the DBMS software are disbursed, if one system goes out, others should still be able to function, and the entire organization is not immobilized.

There are several possible arrangements for connecting the nodes of a distributed system. They may be connected in a star, a ring or a network configuration, as shown in Fig. 1-3. The star configuration is centralized and depends upon the central node for communication among all the nodes. If the central server has problems, the rest of the nodes are inaccessible. In both a ring and a network configuration, the stability of the network does not depend completely upon the stability of any one machine. Particular network issues are beyond the scope of this book.

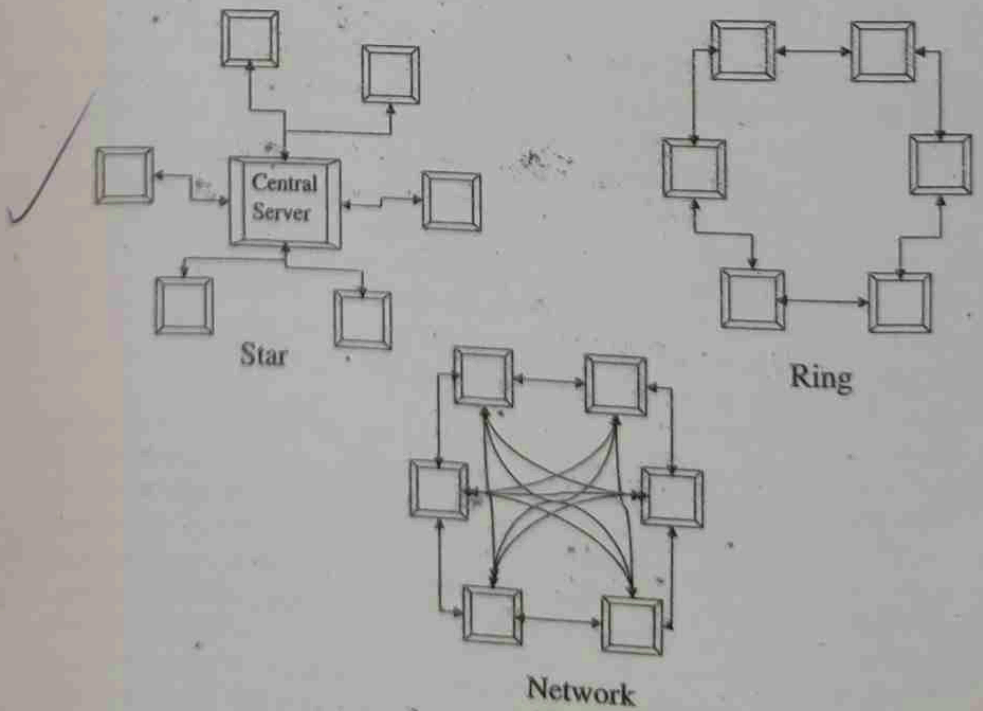


Fig. 1-3. Network configurations.

Example 1.3
Describe how a local medical group of 3 doctors with 2 separate office locations would keep their patient database if it were centralized. How could it be changed to make it a distributed system?

For a centralized system, the entire database would reside on one server at one central location. Client machines at each location would access the central database for patient information. If the central system was down, no patient information could be accessed. In a distributed system, the information for the patients usually seen in location 1 would be kept on a server in that office. Patient information for location 2 patients would be kept on a server in that office. The DBMS would access both locations to find information on any patient. If one of the servers was down, the other could still be accessed.

Example 1.4
Specify whether each system would be single-tiered, two-tiered, or three-tiered.

a. The Happy Nights motel chain allows local managers to purchase a franchise. They can install and use the DBMS of their choice for their reservation system. The only requirement is that they be able to connect and communicate with the central office's system.
b. The Sticky Wicket Company has home offices in Detroit and branches in Chicago and Baltimore. The inventory and parts database is distributed with each branch keeping its own inventory. One central DBMS located in Detroit allows instant ordering of supplies through the central office.

Since both these companies have a number of locations, the systems are obviously not single-tier. The key to the difference is whether there is one central DBMS or whether each local entity runs its own personal database system. Since example (a) allows each franchise to use a different DBMS, middleware will be required to connect these systems with the central office DBMS, resulting in a three-tier system. Because (b) uses one central DBMS with client software at each location, this would be a two-tier system.

1.1.2 WHY WE NEED DBMS

Before continuing with this introduction to DBMS concepts, it is important to specify why we need database management systems. Certainly all readers are aware of the information explosion in today's society. Personal information is stored about each of us in a variety of forms. Anyone who works in any kind of business, either a large or a small organization, knows how important it is to keep accurate records. The advantages of using a DBMS fall into three main categories.

- Proper maintenance of the data
- Providing access to data
- Maintaining security of the data

1.1.2.1 Proper Maintenance of the Data

Proper maintenance of the data will be a recurring theme throughout this book. The users must be able to trust that the data is *accurate* and *up to date*. *Inconsistency* should be avoided and *redundancy* should be minimized. Redundancy occurs when the same information is kept in a variety of places. Inconsistency comes when data are changed in one of those locations and not changed in another. Most database systems provide for *integrity constraints* that must be followed. All these concepts will be considered both explicitly and implicitly in the following chapters. The DBMS is the key to enforcing these characteristics within the database. Each DBMS may manage the data in different ways, but they all are careful to address such data issues.

Example 1.5
In a particular organization, customer names and addresses are kept in one database for the sales department and another database for the billing department. What inconsistency might result from this redundancy?

When a sales person is taking an order, the customer reports a change in address. The sales person might update the record in the sales department. However, when the bill is prepared, it is sent to the old address because the address was not changed in that database.

1.1.2.2 Providing Access to Data

As specified in the previous section, the data are usually shared by a variety of users and programs. Both storage of and access to data should be easy and quick. Concurrent support for all kinds of transactions, both interactive and programmed queries, must be provided by the DBMS. The interactive queries should not have to wait for the application programs to finish. Basically, the data should be accessible precisely when required. It would be unacceptable for the users to wait even a day while the database is updated and checked. The job of the DBMS is to allow for speedy access for all the necessary users while still using proper maintenance procedures.

Another issue surrounding access is the ability to find a particular piece of information from the large amount of data stored. The DBMS must contain flexible methods to access each item in the database while allowing for speedy searches throughout the database to find that item.

1.1.2.3 Maintaining Security of the Data

The DBA is usually the person responsible for the security of the data. Unauthorized access must be prevented, and a variety of levels of permission must be granted to users. Tools are provided for the DBA to enforce all security procedures and meet all the conflicting requirements that arise when many people must access the same database. If two separate users are accessing a

particular table at the same time, the DBMS must not allow them both to make conflicting changes. Such safeguards are part of all systems. The DBMS also provides the tools for easy backup and recovery in case of system failures. Chapter 6 explores a variety of issues surrounding data security.

Example 1.6
Make a list of all the databases you can think of where your name and financial information are kept. How can you check the accuracy of this data?

Information about you is stored by your employer, your school, possibly your religious organization, the government and any banks or credit companies. You can check the accuracy of many of these locations by asking for a credit report or current statement. Many localities have laws requiring schools and government agencies to allow you to see and correct personal information. You will never know if the information is wrong if you don't check it yourself.

It might also be helpful to specify situations when one might not want to use a DBMS. Sometimes these systems result in unnecessary costs when traditional file processing would work just as well. If only one person maintains the data and that person is not skilled in designing a database, the resulting product might also take more time and be less efficient. When you are designing a database, as with any other system, remember that a simple, clear strategy is usually more easily maintained than a complex, confusing design.

1.2 Data Models

Children build model airplanes or model cars as a way of understanding how a real airplane or car is constructed. Through understanding the models, children expect to see a car with four wheels and an airplane with two wings. Real estate developers take prospective customers on tours of a model home to show them how the house is organized and the relationship between the various rooms of the house. Homebuyers can get a good feel for the flow from kitchen to dining room to living room. Models generally allow people to conceptualize an abstract idea more easily.

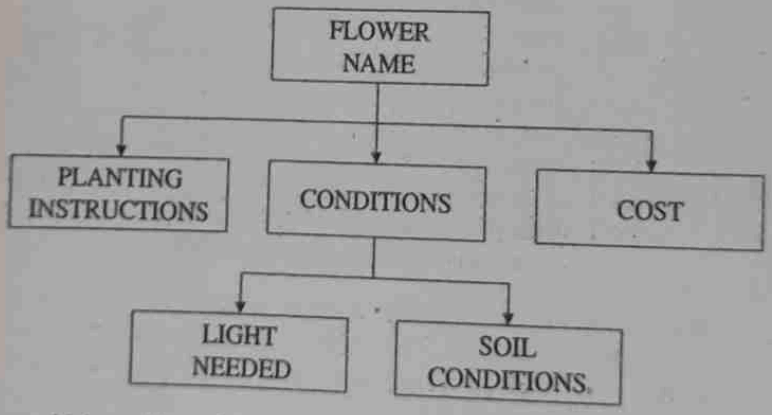
A *data model* is a way of explaining the logical layout of the data and the relationship of various parts to each other and the whole. Different data models have been used throughout the years. In the early years often a *flat file* system, or a simple text file with all the data listed in some order, seemed the easiest. The application program accessed the data usually sequentially for batch processing. Not much interactive access was available. Other models used on big mainframes were the *hierarchical* and the *network* model. The hierarchical database is constructed using a tree model, with one root and several levels of subtrees. Each item has just one link leading to it. Data are accessed beginning with the root and traveling down the tree until the desired details are located. The network model contains many links among the various items of data. Interrelated indices allow access to data from a variety of directions.

logical layout of the data

In 1970, Dr. E. F. Codd described a new kind of model, the *relational* model for database systems.[1] Relational database management systems, where all data are kept in tables or relations, became the new standard. They are much more flexible and easy to use, as almost any item of data can be accessed more quickly than in the other models. Retrieval time is reduced so that interactive access becomes more feasible than in the other models. The use of related tables and views also allows the use of distributed databases that would be difficult in the hierarchical or network models. The data dictionary for the relational model contains the table names, along with column names and data types for each table. In addition, the data dictionary maintains information about all users and privileges. Relational database management systems (RDBMS) will be the model used throughout this book and are explained more fully in Chapter 2.

Example 1.7

Given this model of data for a flower company, would it be hierarchical, network or relational?



This would be a hierarchical model because, though each node can point to several other nodes, each level only has one node pointing to it from above. To find out the amount of light needed, one would need to access first the flower name and then the conditions. It would be difficult to access planting instructions for only those flowers that need to be planted in full sunlight.

Example 1.8

How would the same data be organized in a RDBMS?

In a relational database the data would be kept in a table with one row for each item as shown below. Questions regarding the values in any column are possible any time. For example, the names and planting instructions for each flower that grows in full sunlight would be easily displayed.

[1] E. F Codd, *The Relational Model of Data for Large Shared Data Banks*. This model is further explained in E. F. Codd, *The Relational Model for Database Management Version 2*, Addison-Wesley, Reading, MA, 1990.

Flower Name	Planting Instructions	Conditions	Light Needed	Soil Conditions	Cost

1.3 Database System Architecture

Understanding an abstract model of the data is important in order to describe the architecture of the database system. Recall from earlier in this chapter that one of the major features of databases is that the actual data are separated from the programs that use the data. That fact is important to keep in mind in this entire section which explains database schemas and languages and then describes database system architecture.

Database management systems can also be classified in the way they use their data dictionary. As stated before, the data dictionary contains logical descriptions of the data and its relationships, physical information about data storage, and usually information on users and privileges. Some software vendors also design the data dictionary to keep usage information such as frequencies of queries and other transaction information. Data dictionaries are helpful for all human users, especially the database administrator, as well as invaluable to the application programs and report generators that might access the database.

1.3.1 SCHEMAS AND LANGUAGES

The data model describes the data and the relationships at the abstract level. The *database schema* is used to describe the conceptual organization of the database system. This organization is defined during the design process, usually using the data definition language (*DDL*) provided by the particular software vendor.

The organization of the data can be defined at two levels, logical and physical. The physical organization is related to how the data are actually stored on the disk. The logical organization is the conceptual data model that is being implemented. The DDL allows the user to define the organization of the data at the logical level. The particular DBMS software then takes care of the physical organization of data by mapping from the logical to the physical. In this way, users are protected from having to deal with the hardware level storage of data.

The DDL is used to create the tables and describe the fields within each table. Fig. 1-4 shows the schema diagram for part of a retail store database. The schema shows three tables. Each table contains information about particular objects: customers, orders and employees. The schema contains no information about how the bits of each item are physically organized or exactly where the item is stored on a particular storage device.

ORDER INFORMATION	CUSTOMER INFORMATION	EMPLOYEE INFORMATION
ID	ID	ID
CUSTOMER_ID	NAME	LAST_NAME
DATE_ORDERED	PHONE	FIRST_NAME
DATE_SHIPPED	ADDRESS	USERID
SALES_REP_ID	CITY	START_DATE
TOTAL	STATE	COMMENTS
PAYMENT_TYPE	COUNTRY	MANAGER_ID
ORDER_FILLED	ZIP_CODE	TITLE
	CREDIT_RATING	DEPT_ID
	SALES_REP_ID	SALARY
	REGION_ID	COMMISSION_PCT

Fig. 1-4. Schema diagram for part of retail store database.

It is important to decide on the schema early in the database design process. Once the database has been created and has begun to be populated with data, it is sometimes difficult to change the schema. The actual population of the database with information is accomplished using the data manipulation language (*DML*). The most common language used by many DBMS for this purpose is SQL, which is explained in Chapter 3. The DML allows the user to enter, to retrieve, and to update the data. Some database management systems like Microsoft Access™ allow a graphical interaction with the database, but usually a DML is working in the background.

Example 1.9
Design a possible schema for a doctors' office. The doctors want immediate access to patient medical information. The records clerk needs to be sure all insurance companies are billed and then each patient is billed for the remainder.

One possible simple schema is shown below. The information could be kept in three tables: patient medical history, patient personal information, and insurance company information. The information kept in medical history would be determined by the particular specialty of the doctors. This table would be connected to the personal information through the patient_ID. The insurance company information could be kept in another table connected to each patient by the company_ID. The reader should be aware that this example is not the definition of a complete database. More information would certainly be stored.

PATIENT MEDICAL HISTORY	PATIENT PERSONAL INFORMATION	INSURANCE COMPANY INFORMATION
PATIENT_ID	PATIENT_ID	COMPANY_ID
AGE	NAME	NAME
GENDER	ADDRESS	ADDRESS
PAST_ILLNESS	CITY	CITY
OTHER . . .	STATE	STATE
	ZIP	ZIP
	TOTAL_AMT_DUE	
	INSURANCE_CO_ID	
	AMT_BILLED_INSURANCE	

Example 1.10
Would the user use the DML or the DDL to do each task? (a) Change the customer's address, (b) define an inventory table, (c) enter the information for a new employee.

a. and **c.** Updating a customer address and entering new employee information would be accomplished through the use of the DML. Both these activities entail manipulating data within currently established tables.

b. Defining a new table would entail the use of the DDL. Creating the table and establishing the attributes are part of the data definition.

1.3.2 THREE-LEVEL ARCHITECTURE

The generally accepted method of explaining the architecture of a database system was formalized by a committee in 1975 and more fully explained in 1978.[2]

[2] Dionysios C. Tsichritzis and Anthonly Klug (eds.), *The ANSI/X3/ SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems, Information Systems 3*, 1978.

It is known as ANSI/SPARC architecture, named for the Standards Planning and Requirements Committee of the American National Standards Institute. The three levels are internal, conceptual and external.

- The *internal* level is the one that concerns the way the data are physically stored on the hardware. The internal level is described using the actual bytes and machine-level terminology. Usually the DBMS software takes eare of this level.
- The *conceptual* level, the logical definition of the database, is sometimes referred to as the community view. The data model and the schema diagram are both explanations of the database on the conceptual level. The DBA and assistants maintain the schema and usually are the ones who use the DDL to define the database.
- The *external* level is the one concerned with the users. Whether the users are application programmers or end users, they still have a view, or mental model, of the database and what it contains.

Fig. 1-5 shows a graphical representation of the three levels. The conceptual level, or the community view, is where the physical interior level is interpreted and changed into external views for the users. This figure demonstrates that the DBMS acts as the "go-between" to manage the system by handling interior storage and protecting the users from hardware issues.

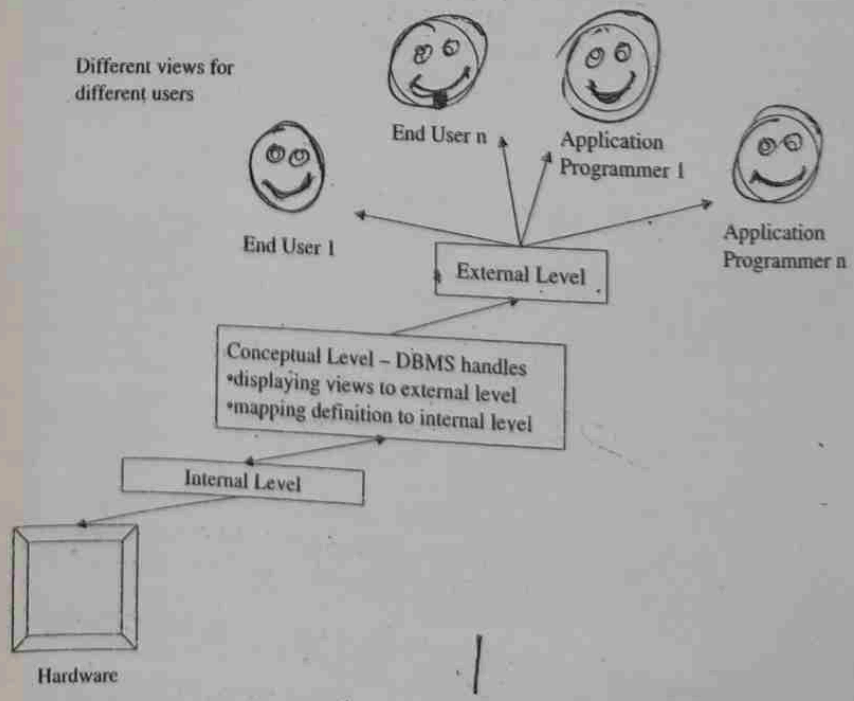


Fig. 1-5. DBMS three-level architecture

To understand the difference between the three levels, consider again the database schema in Fig. 1-4 that describes customers, orders, and employees. That schema would be the conceptual or community view of the database. Particular information is listed for each entity. The internal level would describe exactly which bytes contain the information and how it can be accessed. If User 1 is the payroll clerk, the external view would contain only the employee information. If Application Programmer 1 is designing billing programs, he or she would need all customer and order information as well as information on the particular sales representative in the external view. Fig. 1-6 shows specific information actually available at each level regarding a particular employee.

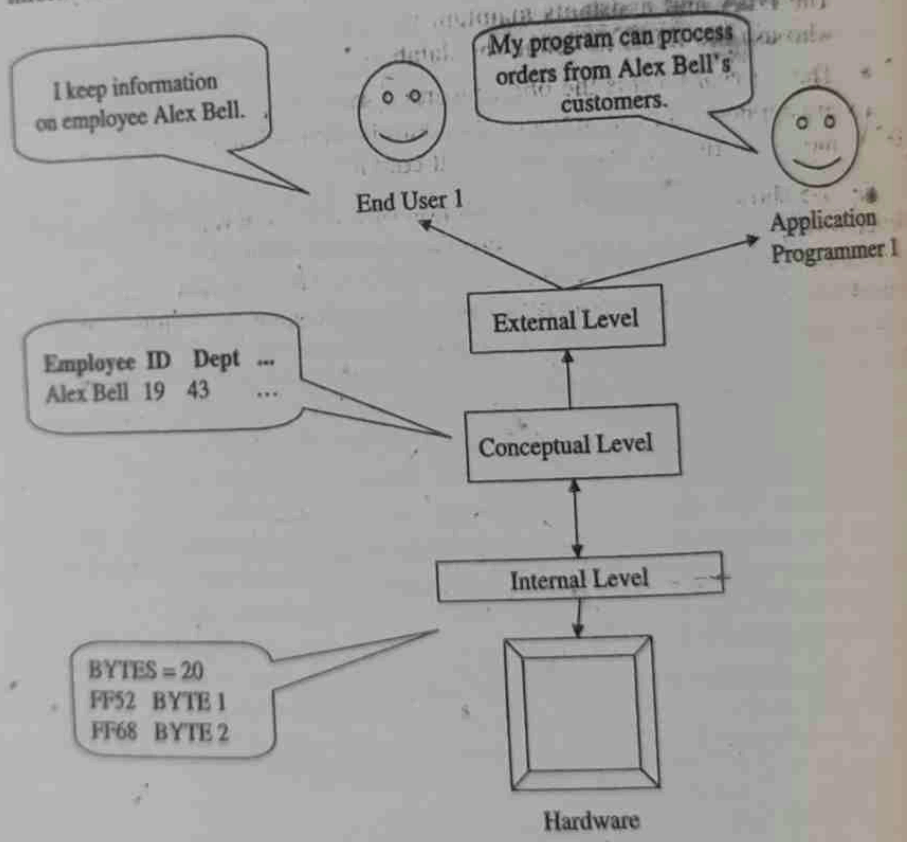


Fig. 1-6. Information at different levels.

Example 1.11

Examine the sample schema in Example 1.9. What would be the internal view, the community view, and the external view of that database?

The tables and the actual contents of each column would be the community view of the database. It might contain such information as

PATIENT_ID	NAME	ADDRESS...
444-44-4444	Sue Jones	345 High St...

The internal view would be the physical location of each element on the disk of the server as well as how many bytes of storage each element needs. The external view would depend upon which user is accessing the database. The doctor would expect to see the patient history. The billing clerk would expect the insurance and billing information.

1.3.3 DATA INDEPENDENCE

The concept of *data independence* is important to address at this time. It has already been stated that the data should be kept separate from the DBMS. At the physical level, the data should be independent of the particular model or architecture. The schema at any of the three levels should be modifiable without interfering with the next higher level. For example, the physical storage of the database might need to be changed. However, this change should not affect either the conceptual view of what is stored or the user's ability to understand and access the data. The data should also be logically independent. Different users and application programs require different information via different logical views. A well-designed system will maintain data independence both physically and logically.

1.3.4 PUTTING THE MODULES TOGETHER

We have discussed all the components of the DBMS. This section focuses on the various software modules usually found in the DBMS and where they can be found with regard to the computer system as a whole. It is probably easiest to approach this environment from the standpoint of the different users explained previously. Fig. 1-7 shows the relationship of users and the various software modules to the actual data.

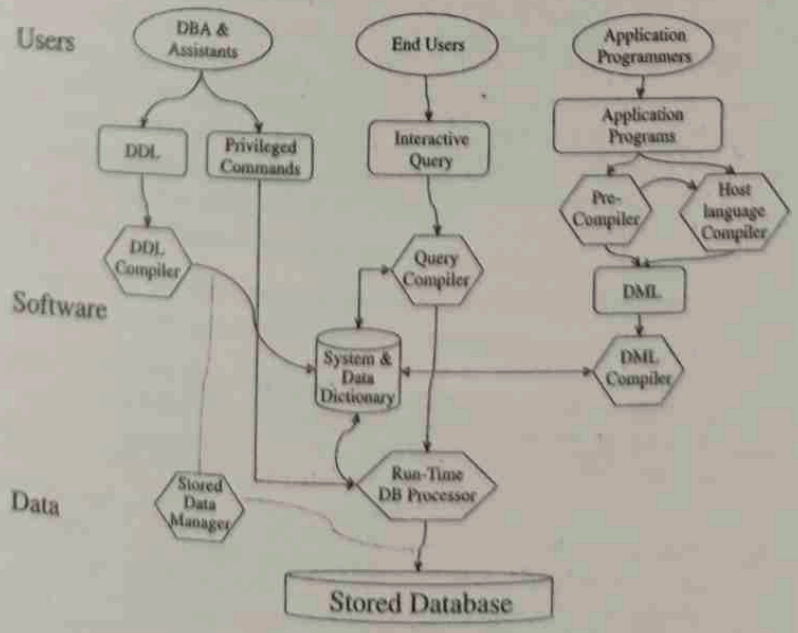


Fig. 1-7. Modules of the DBMS.

The actual data are stored on a disk. The DBA and assistants can issue both privileged commands and DDL statements, which are managed first by the DDL compiler. End users issue interactive queries that are also compiled before processing. The application programmers write the programs that are precompiled to create the DML statements needed, as well as compiled in the host language.

The DBMS provides complete protection for the data through the use of the data dictionary, the run-time database manager, and the stored data manager. Every access to the stored data comes through one or more of these components. The DBMS consistently checks with the data dictionary to be sure all accesses are legal and then processes those commands through the run-time processor. The run-time database manager handles each query, whether retrieval or update. Only the DBA and staff have access to the stored data manager for creation and updating of the actual table structure.

Solved Problems

1.1. Which type of user would usually perform the following functions for an inventory system in a large company?

a. Create a monthly report of current inventory value.
b. Update the number in stock for specific items received in shipment.
c. Cancel the user account for an employee who just retired.
d. Change the structure of the inventory database to include more information on each item.
e. Reply to a phone request regarding the number of a particular item that are currently in stock.

a. Application programmer.
b. End user.
c. DBA or the person on the team designated with that job.
d. DBA or the person on the team designated with that job.
e. End user.

1.2. Consider an inventory database at a kitchen cabinet factory which contains parts information (part number, description, color, size, number in stock, etc.) and vendor information (name, address, purchase order, etc.). Indicate for each user, an accounts payable clerk, a line foreman, and a receiving clerk, which items that user should be able to access and which items the user should be able to access and change.

User	Permission level
a. Accounts payable clerk	Should be able to access and change all data.
b. Line foreman	Needs to access but not change parts information. Probably does not need to have access to any vendor information except maybe name.
c. Receiving clerk	Needs to be able to access and change parts information, such as number in stock. Should be able to access but not change vendor information.

1.3. Why have client/server systems become so prevalent in the business world?

First, the advance of hardware technology allows even small organizations to purchase powerful servers at a reasonable cost. Both easy-to-use software and network resources are also within a reasonable price range. These systems provide a high level of performance while allowing for trusted backup and security capabilities.

1.4. What are some differences in security issues between single-user and multi-user systems?

Single-user systems are often kept secure simply by locking the door to the room containing the computer. Multi-user systems, whether stand-alone or client/server, must employ some kind of password protection allowing different users different levels of access. While backup and recovery issues are similar, having concurrent users results in the necessity for transaction protection, deadlock handling, and locking. These issues are discussed in Chapter 6.

1.5. A county-wide school board wants to create a distributed database for student information. Describe how it might be designed. How would it be different if they wanted a centralized system?

For a distributed system, each local school would keep information on all its students on a server located within the school. All the student databases would be accessible from the central office and from the other schools so that staff in any location could find information on any student. If the school board wanted a centralized system they would need to have one server, probably located at the central office, and then allow access to that database from each local school.

1.6. Specify whether each system would likely be single-tiered, two-tiered, or three-tiered.

a. A woman artist designs and sells jewelry and accessories through mail order or at craft shows. She works out of her home.

b. The school board from the previous example that has a centralized system with each local school accessing data from the server in the central office.

Because the woman in (a) works out of her home, probably she has one computer with the customer and financial information in one DBMS. This would be a single-tier system.

CHAPTER 2

Relational Database Concepts

2.1 Relational Database Management Systems

As indicated in Chapter 1, Database Management Systems are based on data models that allow for a logical or high-level description of the data. A Database Management System based on the relational data model is called a *Relational Database Management System* (RDBMS). In this type of database, the information that comprises the Universe of Discourse is represented as a set of *relations*. In this sense, a RDBMS can then be defined as a collection of relations. Although the notion of a relation can be defined in mathematical terms (see Section 2.2), for all practical purposes, we will represent relations as two-dimensional tables that satisfy certain conditions that will be explained later on. For this reason, in this book we will use the terms tables and relations interchangeably. The reader should keep in mind that in a RDBMS the data is *logically* perceived as tables. That is, tables are logical data structures that we assume hold the data that the database intends to represent. Tables are not physical structures. Each table has a unique name. Tables consist of a given number of *columns* or *attributes*. Every column of a table must have a name and no two columns of the same table may have identical names. The total number of columns or attributes that comprises a table is known as the *degree* of the table. In this book, we use the terms column and attribute interchangeably. The data in the table appears as a set of *rows* or *n-tuples* where n is the number of attributes of the table. Whenever the number of attributes of the table is understood, we can omit the prefix n and refer to the rows of the tables as just rows or tuples. Calling a row an n-tuple (for a fixed n) has the advantage of indicating that the row has n entries (see Example 2.2). However, this terminology is not popular. Most books refer to rows as tuples and vice-versa. In this book, unless otherwise stated we will adhere to the most common terminology. All rows of a table have the same format and represent some object

or relationship in the real world. The total number of rows present in a table *at any one time* is known as the *cardinality* of the table. In legacy systems, the terms *field* and *record* are used as synonyms of the terms attribute and row respectively. Fig. 2-1 shows the general format of the tabular representation of a relation.

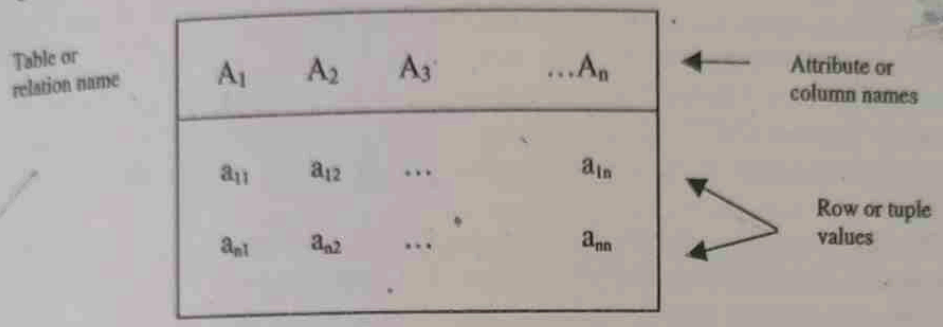


Fig. 2-1. General format of a relation when represented as a table.

We will call the content of a table at any particular point in time a *snapshot* or *instance* of the table. In general, when tables are defined the number of columns remains fixed for the duration of the table. However, the number of rows present in the table is bound to vary since the content of the table reflects the dynamics of the universe that the database intends to capture. New rows may be inserted into the table to represent new facts of the UOD; some rows may be updated to reflect ongoing changes and other rows may be deleted from the table to indicate that some facts are no longer valid or relevant anymore. Notice that tables are required to have at least one column but are not required to have rows. A table with no rows is called an empty table. The process of inserting tuples for the very first time into a table is called *populating the table*.

For *each* column of a table there is a set of possible values called its *domain*. The domain contains all permissible values that can appear under that column. That is, the domain of any column can be viewed as a pool of values from which we can draw values for the column. We will denote the domain of any given column or attribute by *Dom*(*column name*). Observe that any value that appears under a column must belong to its domain. In a table, it is possible that two or more columns may have the same domain. Example 2.1 shows the tabular representation of a relation called EMPLOYEE.

Example 2.1

Given the EMPLOYEE relation shown below, identify its degree and cardinality. For each attribute identify a possible domain.

EMPLOYEE

Id	Last_Name	First_Name	Department	Salary
555294562	Martin	Nicholas	Accounting	55000
397182093	Benakritis	Ben	Marketing	33500
907803123	Adams	Larry	Human Resources	40000

In this table we can identify five attributes or columns: Id, Last_Name, First_Name, Department and Salary. Therefore, the degree of the relation is five.

The table has only three rows or tuples. Therefore, the cardinality of the relation is three.

Possible domains for each of these attributes are as follows:

The domain of the attribute Id, denoted by Dom(Id), is a set of numerical characters. In the United States, for example, it is customary to use the Social Security Number (SSN) of the employee as his or her employee id. Assuming that this is the case in this example, Dom(Id) is the set of nine-digit positive numbers.

The domain of the attribute Last_Name, Dom(Last_Name), is the set of legal last names. In this book we will assume that names consist of a sequence of English alphabetical characters and some other symbols such as a single quote or a hyphen. The number of characters that may comprise a legal last name depends on conventions established by the DBA or the creator of the table. The maximum number of characters typically allowed for names ranges from 20 up to 256 characters.

The domain of the attribute First_Name, Dom(First_Name), is the set of legal first names that a person can have. We will assume that first names follow the same convention used for last names.

For a given company, the domain of the attribute Department, Dom(Department), is the set of names that have been selected as valid department names. In this case, the table shows only three of these values.

The domain of the attribute Salary, Dom(Salary), is a subset of the set of nonnegative real numbers. Notice that a negative salary does not make sense.

In this book, we will assume that every entry of a table or relation has at most a single value.[1] That is, at the intersection of every column and every row there is at most a single value. For any given relation r, for any attribute A of r, and an arbitrary tuple t of r, we will use the notation $t(A)$ to denote the value of the entry of tuple t under the column A. That is, the value at the intersection of column A and row t.

Example 2.2

For the CUSTOMER_ORDER relation shown below, what are the individual values of t(A) if t is an arbitrary tuple and A is an arbitrary attribute of the relation?

CUSTOMER_ORDER

Id	Date_Ordered	Date_Shipped	Payment_Type
1	08/11/1999	08/12/1999	cash
2	08/12/1999	08/12/1999	purchase order
10	08/14/1999	08/15/1999	credit

[1] This guarantees that the relation can be represented in a RDBMS or equivalently that it is in First Normal Form. See Chapter 5, Section 5.2.

If we call t the first tuple shown in the table, we can say that t(Id) = 1, t(Date_Ordered) = 08/11/1999, t(Date_Shipped) = 08/12/1999 and t(Payment_Type) = cash.

If t is the second tuple, we will have that t(Id) = 2, t(Date_Ordered) = 08/12/1999, t(Date_Shipped) = 08/12/1999 and t(Payment_Type) = purchase order.

If t is the third tuple, we will have that t(Id) = 10, t(Date_Ordered) = 08/14/1999, t(Date_Shipped) = 08/15/1999 and t(Payment_Type) = credit.

Notice that the tuples of this relation are 4-tuples. That is, each tuple has 4 entries; one entry for each of the four columns of the table.

From the definitions considered earlier and the notion of a table, whenever a relation is represented by means of a table, we will assume that the following conditions hold:

- The table has a unique name.[2]
- Each column of the table has a unique name. That is, no two columns of the same table may have identical names.
- The order of the columns within the table is irrelevant.
- All rows of the table have the same format and the same number of entries.
- The values under each column belong to the same domain (strings of characters, integer values, real values, etc).
- Every entry (the intersection of a row and a column) of every tuple of the relation must be a single value. That is, no list or collection of values is allowed.
- The order of the rows is irrelevant since they are identified by their content and not by their position within the table.
- No two rows or tuples are identical to each other in all their entries.[3]

2.2 Mathematical Definition of a Relation

In this section we will formalize the definitions of the previous section. Given a finite set of attributes $A_1, A_2, A_3, \ldots, A_n$ we will call a relational scheme R the set formed by all these attributes. That is, $R = \{A_1, A_2, A_3, \ldots, A_n\}$. Associated with each of these attributes there is a nonempty set D_i, $(1 \leq i \leq n)$ called the domain of the attribute A_i and denoted by $Dom(A_i)$. Let D be a new set defined as the union of all the attribute domains. In other words, $D = D_1 \cup D_2 \ldots \cup D_n$. We define a relation r on relational scheme R as a finite set of mappings $\{t_1, t_2, \ldots, t_k\}$ from R to D. The individual mappings t_i are called tuples or n-tuples. For each

[2] This constraint refers to tables within the same tablespace or same database in case of personal databases.

[3] Mathematically, a relation is a set of mappings and therefore there are no duplicate elements. In addition, each record is required to have an attribute whose value uniquely identifies each row.

of these tuples, the value under a particular column A_i, denoted by $t(A_i)$, must be an element of the domain of A_i. That is, if t is any tuple of the relation r then $t(A_i) \in Dom(A_i)$ where the symbol "$\in$" is read "belongs to". If the schema R of a relation r is understood, we will refer to the relation by its name, otherwise we will denote it as r(R). Although other authors prefer to define a relation as a subset of the Cartesian Product of the domains of the attributes of the relation, we have decided to define relations as sets of mappings to avoid any explicit ordering of the attribute names. This corresponds with the tabular representation of the relation since the order of the columns is irrelevant.

2.3 Candidate Key and Primary Key of Relation

The notion of a *key* is a fundamental concept in the relational model because it provides the basic mechanism for retrieving tuples within any table of the database. Formally, given a relation r and its attributes $A_1, A_2, A_3, \ldots, A_n$, we will call any subset $K = \{A_1, A_2, \ldots, A_k\}$ with $(1 \leq k < n)$, of these attributes a *candidate key* if K satisfies the following conditions simultaneously:

(1) For any two distinct tuples t_1 and t_2 of the relation r, there exists an attribute A_j of K such that $t_1(A_j) \neq t_2(A_j)$. This implies that no two different tuples of r will have identical entries in all attributes of K. In other words, *at least one* of the following inequalities will be true $t_1(A_1) \neq t_2(A_1)$, $t_1(A_2) \neq t_2(A_2), \ldots, t_1(A_k) \neq t_2(A_k)$ for any two tuples that we consider in the relation. This condition is known as the *uniqueness property of the key*.[4]

(2) No proper subset K′ of K satisfies the uniqueness property. That is, no element of K can be discarded without destroying the uniqueness property. This condition, called the *minimality property of the key*, guarantees that the number of attributes that comprises the key is minimum.

Since a relation r may have more than one candidate key, one of these candidate keys should be designated as the *primary key* (PK) of the relation. The values of the primary key can then be used as the identification and addressing mechanism of the relation. That is, we will differentiate between the different rows of the relation on the basis of their PK values. We will also uniquely retrieve tuples from a relation based on the values of their PK values. Once a primary key has been selected, the remaining candidate keys, if they exist, are sometimes called *alternate keys*. A RDBMS allows only one primary key per table. A primary key may be composed of one single attribute (*single primary key*) or may be composed of more than one attribute (*composite primary key*). Attributes that are part of any key (primary or alternate) are called *prime*

[4] Some RDBMS allow users to create tables with columns defined with a UNIQUE constraint. This type of constraint forbids duplicate values under any attribute so defined.

attributes. In this book, we will underline the attributes that are part of the primary key. In Example 2.1, attribute Id is underlined because it is the PK of the EMPLOYEE relation. Since Id is the PK of this table no two employees will have identical Id values. Likewise, in Example 2.2, we also underlined the attribute Id of the CUSTOMER_ORDER relation. Since Id is the PK of the CUSTOMER_ORDER relation, no two orders will have the same Id value. Notice that even though these two primary keys (Id of EMPLOYEE and Id of CUSTOMER_ORDER) have identical names they are to be considered different because their underlying meanings are different.

Since the primary key is used to identify uniquely the tuples or rows of a relation, none of its attributes may be *NULL*. This fact imposes an additional condition or *constraint* on the keys known as the *integrity constraint*. In a relation, a NULL value is used to represent missing information, unknown, or inapplicable data. The reader should be aware that a NULL value is not a zero value nor does it represent a particular value within the computer.[5]

Example 2.3

Consider the DEPT table and the rows shown below. Explain whether or not these rows can be inserted into the DEPT table. Notice that DEPARTMENT is the key of the table.

DEPT

DEPARTMENT	NAME	LOCATION	BUDGET
20	Sales	Miami	1700000
10	Marketing	New York	2000000

DEPARTMENT	NAME	LOCATION	BUDGET
10	Research	New York	1500000
	Accounting	Atlanta	1200000
15	Computing	Miami	1500000

10	Research	New York	1500000
	Accounting	Atlanta	1200000
15	Computing	Miami	1500000

No, this row cannot be inserted. It violates the uniqueness property of the key since there is a department 10 already in the table.

No, this row cannot be inserted. It violates the integrity constraint of the key since the department key cannot be NULL.

Yes, this row can be inserted with no problems since no constraint is violated.

[5] In most RDBMS comparisons between nulls are by definition neither true nor false but unknown.

Given a key K of a relation r, we will call a *superkey* of r any set of attributes K' of the relation that contains the key. That is, if K' is a superkey then $K' \subset K$. The symbol "$\subset$" reads "subset of". That is, any set of attributes of the relation r that contains the key is called a superkey. It should be clear that a superkey satisfies the uniqueness property of the key but not the minimality property of the key (see Solved Problem 2.8).

Keys, as we mentioned before, are the basic mechanism for identifying and retrieving the different tuples of a relation. Therefore, when considering the selection of keys we need to choose attributes that satisfy the uniqueness and minimality condition for all permissible data. What this implies is that we cannot select keys for a table based on a list of possible values that may appear in the table. We also need to consider the underlying meaning of the attributes that we are selecting (see Solved Problem 2.7).

Given a relation r, a subset $X = \{A_1, A_2, \ldots, A_k\}$ of attributes of r, and any tuple t of r, we will call the X-value of t, denoted by $t(A_1, A_2, \ldots, A_k)$ the k-tuple $\{t(A_1), t(A_2), \ldots, t(A_k)\}$. That is, the X-value of a tuple t is a k-tuple whose elements are the individual entries at the intersection of the tuple t and the attributes $A_1, A_2, \ldots, A_k$ respectively. If the order of the attributes is understood, the X-value of t can be denoted by t(X).

Primary keys are defined using DDL statements and are automatically enforced by the RDBMS. They are generally defined at the time the tables are created (see Chapter 3, Section 3.2.1).

2.4 Foreign Keys

Because columns that have the same underlying domain can be used to relate tables of a database, the concept of a foreign key allows the DBMS to maintain consistency among the rows of two relations or between the rows of the same relation. This concept can be formally defined as follows: Given two relations r_1 and r_2 of the same database,[6] a set of attributes FK of relation r_1 is said to be a *foreign key* of r_1 (with respect to r_2) if the following two conditions are satisfied simultaneously:

- The attributes of FK have the same underlying domain as a set of attributes of relation r_2 that have been defined as the PK of r_2.[7] The FK is said to reference the PK attribute(s) of the relation r_2.
- The FK-values in any tuple of relation r_1 are either NULL or must appear as the PK-values of a tuple of relation r_2.

From a practical point of view, the foreign key concept ensures that the tuples of relation r_1 that refer to tuples of relation r_2 must refer to tuples of r_2 that already exist. This condition imposed on foreign keys is called the *referential integrity* constraint. Some authors call the table that contains the foreign key a *child table*; the table that contains the referenced attribute or attributes is called

[6] According to this definition of foreign key, relations r_1 and r_2 could be the same relation.

[7] Some DBMS vendors extend this concept to include attributes of r_2 that have been defined as UNIQUE.

concerning data will be given in the next

the *parent table*. Using this terminology, we can say that the FK value in each row of a child table is either null or it must match the PK value of a tuple of the parent table.

Example 2.4

Consider the tables indicated below. Assume that the attribute EMP_DEPT is a FK of the EMPLOYEE table that references the attribute ID of DEPARTMENT. Indicate if the rows shown below can be inserted into the EMPLOYEE table.

ID	NAME	LOCATION
10	Accounting	New York
40	Sales	Miami

DEPARTMENT

EMP_ID	EMP_NAME	EMP_MGR	TITLE	EMP_DEPT
1234	Green		President	40
4567	Gilmore	1234	Senior VP	40
1045	Rose	4567	Director	10
9876	Smith	1045	Accountant	10

EMPLOYEE

9213	Jones	1045	Clerk	30	No, this row cannot be inserted since it violates the referential integrity constraint. There is no department 30 in the department table.
8997	Grace	1234	Secretary	40	Yes, this row can be inserted with no problem since no constraint is violated.
5932	Allen	4567	Clerk	NULL	Yes, this row can be inserted into the table. NULL values are acceptable in the EMP_DEPT column. The NULL value may indicate that the employee has not yet been assigned to a department.

Notice that in the previous example, we use the keyword[8] NULL to indicate explicitly the absence of a department whereas in Example 2.3 the entry for DEPARTMENT was left blank. The reader should be aware that some systems allow both ways to indicate a NULL value whereas some other systems may require that the NULL keyword be used explicitly.

Foreign keys are generally defined after all tables have been created and populated. This avoids problems such as the one illustrated in Solved Problem 2.9 or problems of circularity. The latter problem occurs when one table references values in another table, which in turn, may reference the first table. Integrity constraints are defined using DDL statements and are automatically enforced by the RDBMS.

[8] A keyword is a word that has a specific meaning to the system and cannot be used outside a specific context.

lue in each row
a tuple of the

EPT is a
MENT.
table.

OYEE

rted since it
rity constraint.
n the department

d with no
is violated.

d into the table.
e in the
ULL value may
as not yet been

L to indicate
the entry for
some systems
systems may

created and
ved Problem
n one table
he first table.
utomatically

specific context.

2.5 Relational Operators

We call *relational operators* a set of operators that allow us to manipulate the tables of the database. The entire set of operators that allows us to construct new relations from a set of given relations is called a *relational algebra*. Relational operators are said to satisfy the *closure property* since they operate on one or more relations to produce new relations. When a relational operator is used to manipulate a relation, we say that the operator is "applied" to the relation. To define a relational operator we use *relational calculus*. That is, the set of predicate or truth-value statements that, combined with basic logical operations, determine the shape and membership conditions of elements in the resulting relation. To define the relational operators, we will use the logical symbols: $\exists$ (there exists), $\forall$ (for all), $\vee$ (or), $\wedge$ (and), $\sim$ (not), and some set theory symbols that include $\in$ (belongs to), $\subset$ (subset of), $\varnothing$ (empty set), / or $\ni$ (such that), and some of their negated symbols such as $\notin$ (does not belong to), and $\not\subset$ (not a subset of). Since relational operators are defined by means of equalities, the expression on the left-hand side of the equal sign is the relational algebra expression for the operator. The expression on the right-hand side is the relational calculus definition of the operator. In this section, we will discuss only the Selection, the Projection and the Equijoin relational operators. Some other operations, in particular, Boolean operations on relations will be considered later in the chapter. Each operation on a relation answers a question posed to the database. In database lingo, questions to the database are called *queries*. We will use this term to refer to any relational operation performed on relations with the purpose of retrieving information from the database.

2.5.1 THE SELECTION OPERATOR[9]

This operator, when applied to a relation r, produces another relation whose rows are a subset of the rows of r that have a particular value on a specified attribute. The resulting relation and r have the same attributes. More formally, we can define this operator as follows. Let r be a relation on scheme R, A a specified attribute of r, and a particular element a of the Dom(A). The Selection of r on attribute A for the particular element a is the set of tuples t of the relation r such that t(A) = a. That is, all rows of the new relation—the Selection relation—have a under the column A. The *Selection of r on A* is denoted $\sigma_{A=a}(r)$. Notice that the predicate A = a of $\sigma_{A=a}(r)$ is meant to be understood as t(A) = a. Mathematically, the Selection operator can be defined as follows:

$$\sigma_{A=a}(r) = \{t \in r / t(A) = a\}$$ where the symbol "/" is read "such that".

The scheme of the new relation, $\sigma_{A=a}(r)$ is the same scheme of the relation r.

[9] This operator is also known as the RESTRICT or SELECT operator. We do not use the word SELECT to avoid any confusion with the SQL command that shares the same name.

That is, as was mentioned before, $\sigma_{A=a}(r)$ and r have the same attributes. The following example illustrates how this Selection operator works. Notice that the Selection operator is a *unary operator*. That is, it operates on one relation at a time. From a practical point of view this operator is applied to a relation whenever we are interested in retrieving all possible information about a tuple or set of tuples that have a given value under a particular column.

Example 2.5
Given the EMPLOYEE relation of the previous example, find all the information contained in the table for all employees who work for department 10.

Since we need to retrieve *all* the information about the employees who work for department 10, it is necessary to determine $\sigma_{EMP\_DEPT=10}(EMPLOYEE)$. Notice that in this example the condition that needs to be satisfied by the tuples of the EMPLOYEE relation to appear as tuples of the new SELECTION relation is t(EMP_DEPT) = 10. The resulting table is shown below.

$\sigma_{EMP\_DEPT=10}(EMPLOYEE)$

EMP_ID	EMP_NAME	EMP_MGR	TITLE	EMP_DEPT
1045	Carson	4567	Director	10
9876	Smith	1045	Accountant	10

Notice that all rows of this new relation have the value of 10 under the column EMP_DPET

2.5.2 THE PROJECTION OPERATOR

The Projection operator is also a unary operator. Whereas the Selection operator chooses a subset of the rows of the relation, the Projection operator chooses a subset of the columns. This operator can be formally defined as follows. The *Projection of relation r onto a set X of its attributes*, denoted by $\pi_X(r)$, is a new relation that we can obtain by first eliminating the columns of r not indicated in X and then removing any duplicate tuple. The attributes of X are the columns of the Projection relation. Mathematically, the Projection relation can be defined as follows: Let r be a relation with relational scheme R and let $X = \{A_1, A_2, \ldots, A_k\}$ be a subset of its attributes, then $\pi_X(r) = \{t(X)/t \in r\}$. Notice that entries of the rows of the Projection relation are formed by taking from each of the tuples t of r the entries corresponding to the attributes defined in X. That is, the entries for a tuple j of $\pi_X(r)$ are formed by selecting the entries $t_j(A_1), t_j(A_2), \ldots, t_j(A_k)$ from the tuple j of the relation r. Since relations have been defined as sets it is necessary to eliminate all duplicate tuples from $\pi_X(r)$. From a practical point of view, this operator is applied whenever we are interested in the different values currently present under a particular column or

the different combinations of values currently present in two or more columns. The following example illustrates the use of the Projection operator.

Example 2.6
Using the DEPARTMENT table shown below, what are the locations of the different departments? From the answer to the previous question, can we tell the total number of locations that are currently present in the DEPARTMENT table?

DEPARTMENT

ID	NAME	LOCATION
10	Accounting	New York
30	Computing	New York
50	Marketing	Los Angeles
60	Manufacturing	Miami
90	Sales	Miami

Since we need to determine the different values that are currently present in the LOCATION column, we need to find the Projection of the DEPARTMENT table on the attribute LOCATION. That is, we need to find $\pi_{LOCATION}$(DEPARTMENT). In this case, r = DEPARTMENT and X = {LOCATION}. The tuples of the Projection are obtained by retrieving the value t(LOCATION) for each of the tuples of the relation DEPARTMENT. Since two of the tuples have New York as their locations, only one of these values will appear in the projection. Similar argument can be made for the tuples that have Miami as their locations. The resulting relation is shown next.

$\pi_{LOCATION}$(DEPARTMENT)

LOCATION
New York
Los Angeles
Miami

Observe that LOCATION is the sole attribute of this table.

In general, the projection of this relation on the attribute LOCATION cannot give us the total number of locations present in the DEPARTMENT table since duplicate values are eliminated. Notice that the Projection relation only has three locations whereas the DEPARTMENT relation has a total of five locations.

Example 2.7
Using the table of the previous example, what are the different departments and their locations?

In this case, X = {NAME, LOCATION} and r = DEPARTMENT. The resulting relation is shown below.

$\pi_{NAME, LOCATION}(DEPARTMENT)$

NAME	LOCATION
Accounting	New York
Computing	New York
Marketing	Los Angeles
Manufacturing	Miami
Sales	Miami

The resulting relation does not contain duplicate values since the different combinations of name and location are unique. Notice that the tuples (Manufacturing, Miami) and (Sales, Miami) are different. Likewise (Accounting, New York) and (Computing, New York) are considered to be two different tuples.

Before considering the next operator, it is necessary to define the *concatenation operator*. This operator is a tuple operator instead of a table operator. Given two tuples $s = (s_1, s_2, \ldots, s_n)$ and $r = (r_1, r_2, \ldots, r_m)$, the concatenation of r and s is the (m + n)-tuple defined as follows:

$\overline{rs} = (s_1, s_2, \ldots s_n, r_1, r_2, \ldots r_m)$ where the symbol $\overline{rs}$ is read the "concatenation" of tuples r and s. Notice that the number of entries in the tuples r and s are not necessarily the same. For example, if r = (a, b, c) and s = (1,2) then $\overline{rs}$ = (a, b, c, 1, 2).

2.5.3 THE EQUIJOIN OPERATOR

The Equijoin[10] operator is a binary operator for combining two relations not necessarily different. In general, this operator combines two relations on all their common attributes. That is, the join consists of all the tuples resulting from concatenating the tuples of the first relation with the tuples of the second relation that have identical values for a common set of attributes X. By common attributes we mean attributes that, although they may not have the same name, must have the same domain and underlying meaning. Mathematically, this definition can be expressed as follows: Let r be a relation with a set of attributes R and let s be another relation with a set of attributes S. In addition, let us

[10] This type of join is also called a natural join.

assume that R and S have some common attributes and let X be that set of common attributes. That is, $R \cap S = X$. The join of r and s, denoted by $r\,Join\,s$, is a new relation whose attributes are the elements of $R \cup S$. In addition, for *every tuple* t of the $r\,Join\,s$ relation, the following three conditions must be satisfied simultaneously: (1) $t(R) = t_r$ for some tuple t_r of the relation r (2) $t(S) = t_s$ for some tuple t_s of the relation and (3) $t_s(X) = t_r(X)$.

An equivalent definition of the Join operator is as follows:

$$r\,Join\,s = \{rs / s \in r \text{ and } s \in s \text{ and } r(R \cap S) = s(R \cap S)\}$$

This definition and the previous one are equivalent. The next example illustrates how this operator works.

Example 2.8

Join the tables shown below on their common attributes DEPARTMENT and NAME. Write a possible user's query that can be satisfied by the result of this operation. Can these two tables be joined on the attribute ID?

DEPARTMENT

ID	NAME	LOCATION
100	Accounting	Miami
200	Marketing	New York
300	Sales	Miami

EMPLOYEE

ID	NAME	DEPT	TITLE
100	Smith	Sales	Clerk
200	Jones	Marketing	Clerk
300	Martin	Accounting	Clerk
400	Bell	Accounting	Sr. Accountant

In this case, the common attributes are the DEPT and NAME attributes. The join of these two tables, denoted by DEPARTMENT *Join* EMPLOYEE, is shown below. Since both tables have an attribute called ID, to avoid confusing the attribute ID of the DEPARTMENT table with the attribute ID of the EMPLOYEE table, it is necessary to qualify each attribute by preceding it with its corresponding table name before joining the tables. For a similar reason, the attribute NAME of the EMPLOYEE and DEPARTMENT tables has to be qualified. Observe that this is consistent with the requirement that in any table the column names must be different. Notice that the common column was not duplicated. The results of this join operation could be used to satisfy a user's request to "display all the information about the employees along with their department's id, name and location".

DEPARTMENT *Join* EMPLOYEE

DEPT ID	DEPARTMENT NAME	LOCATION	EMPLOYEE ID	EMPLOYEE NAME	TITLE
100	Accounting	Miami	300	Martin	Clerk
100	Accounting	Miami	400	Bell	Sr. Accountant
200	Marketing	New York	200	Jones	Clerk
300	Sales	Miami	100	Smith	Clerk

In this Join operation notice that the common attribute is DEPARTMENT NAME. That is, X = {DEPARTMENT}. In addition, observe that this join satisfies the three conditions mentioned above. In fact, if r is the DEPARTMENT relation and s is the EMPLOYEE relation we will have that their respective schemes are:

$$R = \{ID, NAME, LOCATION\} \quad \text{and}$$
$$S = \{ID, NAME, DEPT, TITLE\}$$

Observe also that for every tuple t of the Join relation we must have

$t(R) = t(ID, NAME, LOCATION) = t_r$ for some tuple of r.

$t(S) = t(ID, NAME, TITLE) = t_s$ for some tuple of s.

$t_r(X) = t_s(X)$

If we include the attribute names for sake of the explanation and consider the first tuple of the Join relation we have that

DEPT ID	DEPARTMENT NAME	LOCATION	EMPLOYEE ID	EMPLOYEE NAME	TITLE
100	Accounting	Miami	300	Martin	Clerk

If we call t this first tuple notice that

t(DEPARTMENT ID, DEPARTMENT NAME, LOCATION) =
(100, Accounting, Miami) = t_r

t(EMPLOYEE ID, EMPLOYEE NAME, DEPT, TITLE) =
(300, Martin, Accounting, Clerk) = t_s

t_r(DEPARTMENT) = t_s(DEPARTMENT) = Accounting

The reader can verify that the three conditions indicated above for the Join are satisfied by the remaining tuples of the relation. We leave this as an exercise for the reader.

The tables DEPARTMENT and EMPLOYEE *cannot* be joined on the attribute ID because the attribute ID of the EMPLOYEE table and the attribute ID of the DEPARTMENT table have different meanings. One is an employee's id while the other is a department's id. This illustrates the fact that two tables cannot be joined just because they have attributes with the same name. To join two tables on their common attributes, these attributes need to have same domain and the same underlying meaning.

In the database literature, there are several types of joins. The equijoin or natural join previously defined requires that the condition to be satisfied by the tuples of the intervening relations be equality. A *composition* is a natural join with the common join attribute or attributes deleted. A *theta join* is a join where the condition that needs to be satisfied by the tuples may be defined by means of one the following logical operators $\leq$, $\geq$, $\neq$, $<$, or $>$. In general, the latter operation is not directly supported by the RDBMS manufacturers but it can be implemented as a combination of Selection and Projection operations (see Solved Problem 2.17).

2.6 Set Operations on Relations

Since relations have been defined as mathematical sets, the basic binary operations that are traditionally considered as "set operations" can also be applied to relations. These operations are: Union, Intersection, Difference and the Cartesian product. Of all these operations the union is perhaps the most powerful and interesting. The first three of these operations require that the attributes of the participating relations be *union compatible*. We will say that two sets of attributes A and B are union compatible if they are of the same degree and the corresponding domains are of the same data type. Notice that this definition does not require that the names of the attributes be the same. In practical terms the purpose of all these operations is to allow the result of the execution of multiple operations to be displayed as a single statement.

As Chapter 3 will show, all these operations are implemented using variations of the SQL statement SELECT. Since the result of each of these operations is a single table and the attributes of the participating tables are not required to be the same, what will be the names of the columns of the "final" table? In this book, we will follow the convention that the result table will have the column names of the "first" table. This will be clarified shortly.

2.6.1 UNION

Given two relations r(R) and s(S) with union compatible schemes, the UNION of these two relations, denoted by r ∪ s, is the set of all tuples that are currently present in r or are currently present in s or are present in both relations. In mathematical terms, the UNION can be defined as follows:

$r \cup s = \{t/t \in r \vee t \in s\}$ where the symbol "∪" is read "union" and the symbol ∨ is read "or".

Like in any other relation, there are no duplicate tuples in the UNION relation. The scheme of the UNION, that is, its set of attributes is the scheme of the relation r since this is the relation that is named first in the UNION operation. Observe that if we had written s ∪ r the attributes of s would have appeared as column headings of the resulting relation.

Example 2.9

Given the relations shown below, find the union of these two relations. What query will this operation answer?

C_PROGRAMMER

Employee_Id	Last_Name	First_Name	Project	Department
101123456	Venable	Mark	E-commerce	Sales Department
103705430	Cordani	John	Firewall	Information Technology
101936822	Serrano	Areant	E-commerce	Sales Department

JAVA_PROGRAMMER

Employee_Id	Last_Name	First_Name	Project	Department
101799332	Barnes	James	Web Application	Information Technology
101936822	Serrano	Areant	E-commerce	Sales Department

C_PROGRAMMER ∪ JAVA_PROGRAMMER

Employee_Id	Last_Name	First_Name	Project	Department
101123456	Venable	Mark	E-commerce	Sales Department
103705430	Cordani	John	Firewall	Information Technology
101799332	Barnes	James	Web Application	Information Technology
101936822	Serrano	Areant	E-commerce	Sales Department

Notice that there are no duplicate tuples in the resulting relation. Observe also that it is not possible by mere examination of the resulting relation to tell which rows were selected from which table. This operation may answer a query that requests information about all employees that are C programmers or Java programmers or both.

The UNION is a commutative operation. That is, for any two relations r and s, we have that $r \cup s = s \cup r$. The observant reader should realize that this is a direct consequence of the commutativity of the Boolean *OR* operator.

2.6.2 INTERSECTION

Given two relations r(R) and s(S) with union compatible schemes, the intersection of these two relations, denoted by $r \cap s$, is the set of tuples currently present in both relations. That is, the set of tuples common to both relations. In mathematical terms, the INTERSECTION can be defined as follows:

$r \cap s \Leftarrow \{t/t \in r \wedge t \in s\}$ where the symbol "$\cap$" is read "intersection" and the symbol "$\wedge$" is read "and". Like in any relation, there are no duplicate tuples in the INTERSECTION relation.

Example 2.10
Given the tables of the previous example, find the intersection of the given relations. What query will this operation answer?

The tuples of the intersection of these two relations are the tuples that are currently present in both relations. In this case there is only one tuple common to both relations. This operation will answer a query that requests information about all employees that are both C and Java programmers.

C_PROGRAMMER ∩ JAVA_PROGRAMMER

Employee_Id	Last_Name	First_Name	Project	Department
101936822	Serrano	Areant	E-commerce	Sales Department

The INTERSECTION is a commutative operation. That is, $r \cap s = s \cap r$ for any two relations r and s. This is direct consequence of the commutativity of the Boolean *AND* operator.

2.6.3 DIFFERENCE

Given two relations r(R) and s(S) with union compatible schemes, the DIFFERENCE of relations R and S (in that order), denoted by r − s, is the set of tuples that are currently present in r and are not currently present in s. The symbol can be read as a "difference" or as a "minus". In mathematical terms, the difference of these two relations can be defined as follows:

$r - s = \{t/t \in r \wedge t \notin s\}$ where the symbol "$\notin$" is read "does not belong to" and the symbol "$\wedge$", as we indicated before, is read "and".

Example 2.11
Find the difference of the relations C_PROGRAMMER – JAVA_PROGRAMMER and JAVA_PROGRAMMER – C_PROGRAMMER. What queries do these operations answer? Use the relations defined in Example 2.10.

The relation C_PROGRAMMER – JAVA_PROGRAMMER contains the tuples that are currently present in the relation C_PROGRAMMER and are *not* currently present in the relation JAVA_PROGRAMMER. The Difference relation is shown below.

C_PROGRAMMER – JAVA_PROGRAMMER

Employee_Id	Last_Name	First_Name	Project	Department
101123456	Venable	Mark	E-commerce	Sales Department
103705430	Cordani	John	Firewall	Information Technology

This query may answer a request to find all C programmers that are not Java programmers.

The relation JAVA_PROGRAMMER – C_PROGRAMMER contains the tuples that are currently present in the relation JAVA_PROGRAMMER and are not currently present in the relation C_PROGRAMMER. The difference relation is shown below.

JAVA_PROGRAMMER – C_PROGRAMMER

Employee_Id	Last_Name	First_Name	Project	Department
101799332	Barnes	James	Web Application	Information Technology

This query may answer a request to find all Java programmers that are not C programmers.

Notice that the order in which the relations are named in the difference operation is important. In general, and as this example illustrates, given two relations r and s, we have that $r - s \neq s - r$. This result shows that the DIFFERENCE of relations is not a commutative operation.

2.6.4 CARTESIAN PRODUCT[11]

Given two nonempty relations r(R) and s(S), the CARTESIAN PRODUCT of these two relations, denoted by $r \otimes s$, is defined as the relation whose tuples are formed by concatenating every tuple of relation r with every tuple of relation s. In mathematical terms, the Cartesian product is defined as follows:

$r \otimes s = \{\overline{pq} / p \in r \text{ and } q \in s\}$ where, as indicated before, the symbol $\overline{pq}$ reads "the concatenation of tuples p and q".

The degree of the CARTESIAN PRODUCT relation is the sum of the degrees of the original relations with duplicate names qualified if necessary. The cardinality of the CARTESIAN PRODUCT is the product of the cardinality of the original relations. The reader should be careful when applying the Cartesian product because the result relation may make no sense. This operation is sometimes useful if followed by a SELECTION operation that matches values of attributes coming from the component relations (see Solved Problem 2.16). Of all the set operations on relations the Cartesian product is the most time consuming. Therefore, we caution the reader in the use of this operation.

Example 2.12

Given the relations shown below, find the Cartesian product of these two relations.

BUYER

ID_NUMBER	ITEM
100	A
234	B
543	C

[11] This operation is also called the cross product.

PRODUCT

CODE	NAME	PRICE
A	Bike	250
B	Spikes	90
C	Goggles	15
D	Gloves	35

The Cartesian product of these two relations is shown below. As the resulting relation shows, this Cartesian product has 5 attributes = 2 (from BUYER) + 3 (from PRODUCT) and 12 tuples = 3 (from BUYER) * 4 (from PRODUCT). The Cartesian product is shown below.

ID_NUMBER	ITEM	CODE	NAME	PRICE
100	A	A	Bike	250
100	A	B	Spikes	90
100	A	C	Goggles	15
100	A	D	Gloves	35
234	B	A	Bike	250
234	B	B	Spikes	90
234	B	C	Goggles	15
234	B	D	Gloves	35
543	C	A	Bike	250
543	C	B	Spikes	90
543	C	C	Goggles	15
543	C	D	Gloves	35

Notice that this relation, besides containing tuples that can provide us with some useful information, also contains tuples that are meaningless. For instance, from the first tuple we can see that customer number 100 bought a bike for $250. However, what is the meaning of the second tuple?

2.7 Insertion, Deletion and Update Operations on Relations

As indicated before in Section 2.1, the contents of relations are, in general, time-varying. Some of the most common operations applied to relations that allow them to change over time are: Insertion, Deletion and Update operations. Although these operations can be formally defined in mathematical terms, their definitions are too complicated to be of any practical use. Therefore, to define them we will use a less formal approach that allows us to visualize the effect of the operations.

2.7.1 INSERTING A TUPLE INTO A TABLE

Given a relation r with relation scheme $R = \{A_1, A_2, A_3, \ldots, A_n\}$. The format of the INSERT operation is as follows:

INSERT INTO *relation-name* $(A_1 = v_1, A_2 = v_2, A_3 = v_3, \ldots, A_n = v_n)$

where the values $v_1, v_2, \ldots, v_n$ belong to the domain of the attributes $A_1, A_2, A_3, \ldots, A_n$ respectively. If the order of the attributes is understood, we will write the INSERT operation as

INSERT INTO *relation-name* $(v_1, v_2, v_3, \ldots, v_n)$

The effect of this operation, as its name indicates, is to add a new tuple t to the relation where $t(A_i) = v_i$.

Notice that the effect of this operation is not guaranteed to succeed; the INSERT operation may fail for one of the following reasons:

- A given value may not belong to the domain of its corresponding column. That is, $v_i \notin \text{Dom}(A_i)$.
- The tuple that is being inserted does not have the appropriate number of entries as determined by the scheme of the relation.
- The key value of the tuple that is being inserted may duplicate the value of the PK of a tuple already present in the relation.
- One or more values of the tuple being inserted may duplicate the values under one or more columns that have been defined as UNIQUE.
- The name of the relation does not exist in the database, or the attributes as mentioned in the INSERT INTO operation, do not exist in the relation.

In all cases where a violation occurs the system will issue an appropriate error message. The nature of the message depends on the RDBMS being used. The following example illustrates the use of the INSERT operation.

Example 2.13

Given the PATIENT-ACCOUNT relation shown below, state whether or not the indicated tuples can be inserted into the relation. Assume that character data needs to be enclosed in double quotes within the INSERT operator.

PATIENT-ACCOUNT (ACCOUNT, AMOUNT-DUE, DEPARTMENT, DOCTOR-ID, TREATMENT-CODE)

where the corresponding domains are:

Dom(ACCOUNT) = character string, length 9 characters.
DOM(AMOUNT-DUE) = numeric, maximum 9 digits with two optional decimals.
Dom(DEPARTMENT) = {G, T, L, X-rays, I}
Dom(DOCTOR-ID) = character, 4 digits maximum, range from 1000 through 2000.
Dom(TREATMENT-CODE) = {G100, G110, G120, T130, L140, L150, X160, X170, I180, I190, I200}

PATIENT-ACCOUNT

ACCOUNT	AMOUNT-DUE	DEPARTMENT	DOCTOR-ID	TREATMENT-CODE
980543990	2456	G	1200	X160
804804308	245.45	T	1123	G100
173402644	589.25	L	1111	I180

a. INSERT INTO patient-account (ACCOUNT = "890345255", AMOUNT-DUE = 256, DEPARTMENT = "G", DOCTOR-ID = "1500", TREATMENT-CODE = "T130")

This tuple can be inserted with no problems since it does not violate any of the restrictions indicated above.

b. INSERT INTO patient-account (ACCOUNT = "980543990", AMOUNT-DUE = 256, DEPARTMENT = "G", DOCTOR-ID = "1500", TREATMENT-CODE = "T130")

This tuple cannot be inserted because it duplicates the primary key of one of the existing tuples.

c. INSERT INTO patient-account (ACCOUNT = "890345255", AMOUNT-DUE = 256, DEPARTMENT = "G", DOCTOR-ID = "1500")

This tuple cannot be inserted because it does not have the appropriate number of entries.

d. INSERT INTO patient-account (ACCOUNT = "564890155", AMOUNT-DUE = 256, DEPARTMENT = "G", DOCTOR-ID = "1500", TREATMENT-CODE = "T130")

This tuple cannot be inserted because one of the columns is not recognized by the system. Column AMOUNT_DUE has been misspelled.

e. INSERT INTO patient-account (ACCOUNT = "721307804", AMOUNT-DUE = 256, DEPARTMENT = "G", DOCTOR-ID = "3500", TREATMENT-CODE = "T130")

This tuple cannot be inserted because the value for the DOCTOR-ID attribute is not within the acceptable range.

2.7.2 DELETING A TUPLE FROM A TABLE

The DELETE operation removes a particular tuple from a relation. To remove a specific tuple it is necessary to identify it by the attributes of its primary key. Given a relation r with relation scheme $R = \{A_1, A_2, A_3, \ldots, A_n\}$. The format of the DELETE operation is as follows:

DELETE FROM *relation-name* WHERE *search-condition*

The search-condition generally specifies the values of the key attributes of the particular tuple that we want to delete. However, whenever more than one tuple is to be deleted by the same operation, the search condition may be specified by the value of any other attribute. In this book, we will indicate the search-condition as $(A_1 = v_1, A_2 = v_2, A_3 = v_3, \ldots, A_k = v_k)$ where the attributes $A_1, A_2, A_3, \ldots, A_k$ comprise the key of the relation. If the attributes are understood, we will list only the v_i values. The DELETE operation fails if one of the following conditions occurs:

- The indicated tuple is not present in the relation.
- The tuple to be deleted is referenced by a foreign key of another relation. In this case, we say that the operation violates the integrity constraint.
- The relation does not exist or the tuple does not conform to the scheme of the relation.

Deleting the last tuple of a relation is permissible since the empty relation is allowed. That is, a relation with no tuples is legal. In all cases where a violation occurs the system will issue an appropriate error message.

Example 2.14

Using the PATIENT-ACCOUNT relation of the previous example, indicate whether or not it is possible to delete the tuples indicated below.

a. DELETE FROM patient-account WHERE account = "980543990"

This operation successfully removes the tuple whose primary key is 980543990.

b. DELETE FROM patient-account WHERE account = 980543990

This operation fails because the value of the account attribute is not found. Notice that we have made the assumption that character data needs to be enclosed in double quotes. In this case, the RDBMS is looking for a numerical value instead of a character value. Some systems automatically translate character data into numeric data and vice versa. However, the reader should not rely on automatic translations. In this book, we will assume that no automatic translation occurs.

c. DELETE FROM patient-account WHERE accounts = "980543990"

This operation fails because the RDBMS does not recognize the attribute accounts. Notice that the key attribute is singular and not plural.

2.7.3 UPDATING A TUPLE OF A TABLE

To update a tuple is to change the values of one or more of its attributes. The tuple that we want to update needs to be identified by its key or some other search attribute. The format of this operator is as follows:

UPDATE *table-name* SET *column_name* = *new-value* WHERE *search-condition*

The reader should be aware that the UPDATE operator is just a convenience offered by the vendors of the RDBMS since we could delete the tuple that we want to change and then add a new tuple with the correct values. The reasons that this operator may fail are the same reasons that the INSERT and DELETE operations may fail.

Example 2.15

Update the PATIENT-ACCOUNT table to show that patient with account 804804308 has made a payment of $45.45 and now owes $200.00. The corresponding UPDATE operation is as follows:

UPDATE patient-account SET amount-due = 200 WHERE account = "804804308"

This operation successfully sets the value of the attribute amount-due to its new value of 200.

2.8 Attribute Domains and Their Implementations

As indicated before, the domain of an attribute defines the characteristics of the values that a table column may contain. In any RDBMS the domain of any given attribute is implemented using a data type. National and international organizations such as the ANSI (American National Standards Institute) and the ISO (International Standard Organization) have defined a set of basic data types. These data types, although supported by most of the RDBMS vendors, have implementation details that vary from vendor to vendor. Table 2-1 shows some of the basic SQL data types and their implementations for selected RDBMS vendors.

Table 2-1. Some standard SQL data types and some of the RDBMS vendors' implementations

STANDARD SQL	ORACLE	ACCESS	DB2
Character(n) n is number of characters.	*Char(n)* fixed length with up to 255 characters maximum.	*Text* fixed length with up to 255 characters maximum.	*Character(n)* same as ORACLE Char(n).
Character varying (n) n characters. Storage fits the size of content.	*Varchar2(n)* varying length with up to 2000 characters maximum.	*Text* varying length up to 255 characters max or *Memo* varying length up to 64,000 characters maximum.	*Varchar(n)* same as ORACLE varchar2(n).
Float (p), where *p* is total number of digits.	*Number* ranges from 1.0×10^{-130} to 38 9s followed by 88 0s.	*Single or Double* depending on range of data values.	*Float* same as ORACLE NUMBER.
Decimal(p,s) at least *p* digits with *s* defined by vendor.	*Number(p,s)* *p* ranges from 1 to 38, whereas *s* ranges from −84 to 127.	*Integer or Long Integer* depending on range of data values.	*Integer* same as ORACLE NUMBER(38).

Columns of data type *character(n)* or *character varying(n)*, where n is the maximum number of characters that can be stored in the column, are generally used for data containing text or numbers that are not involved in calculations. Examples of this data type are names, addresses, employee identification numbers, social security numbers, and telephone numbers. The primary difference between the character(n) and character varying(n) data types is how they store strings (sequence of characters) shorter than the maximum column length. When a string with fewer than n characters is stored in a character(n) column, the RDBMS pads blank spaces to the end of the string to create a string that has exactly n characters. When a string with fewer than n characters is stored in a character varying(n) column, the RDBMS stores the string "as is" and does not pad it with blank spaces. For this reason, if we know that the contents of a character column may vary in length, it is better to define the column as character varying since the RDBMS can store this information more efficiently. Text data, whether it is stored as fixed or varying type, is always case sensitive. For example, the string 'abc' is different than the string 'aBc'. Any embedded blank counts as a character. For instance, 'abc' and ' abc ' are different strings since the latter contains at least one embedded blank.

Columns of type *float(n)*, where n is the total numbers of digits, are generally used to represent large numerical quantities or scientific computations. For example, in Oracle we can use float data types to represent numbers in the range between 1.0×10^{-10} and $1.0 \times 10^{+10}$.

Columns of type *decimal(p,s)* are used to represent fixed-point numbers. The *precision*, p, is the total number of digits both to the right and to the left of the decimal point. The *scale*, s, is the number of decimal digits to the right of the decimal point. When the number we are representing is a whole number, the scale is set equal to zero. An example of this data type is the number 123.23 which can be specified as decimal(5,2). Whole numbers such as 125 can be specified as decimal(3,0).

Although not shown on Table 2-1, another data type that is commonly used by all RDBMS is *DATE*. This data type allows users to store date and time information. Date is generally displayed in the default format DD-MM-YY where DD stands for Day, MM stands for month and YY stands for year.

CHAPTER 3

An Introduction to SQL

3.1 Introduction to SQL Language

The previous chapter detailed the theoretical and mathematical background for creating, maintaining and retrieving items from tables in relational databases. This chapter presents a general introduction to how these tasks are performed for particular RDBMS systems. We will present only an overview of SQL. For more detailed information on the capabilities of this language, consult other sources that specifically deal with SQL as a language.[1]

SQL is the standard computer language used to communicate with relational database management systems. The SQL standard has been defined by the American National Standard Institute (ANSI) and the International Standards Organization (ISO). The official name of the language is International Standard Database Language SQL (1992). The latest version of this standard is commonly referred to as SQL/92 or SQL2. In this book, we will refer to this standard as the ANSI/ISO SQL standard or just the SQL standard.

The ORACLE Corporation, formerly Relational Software Inc, produced the first commercial implementation of the language in 1979. Although most relational database vendors support SQL/92, compliance with the standard is not 100%. Currently, there exist several flavors of SQL on the market since each RDBMS vendor tries to extend the standard to increase the commercial appeal of its product. In this chapter, we adhere to the SQL/92 standard whenever possible. However, we will illustrate the implementations of these features using

[1] One such book is *Schaum's Outline: Fundamentals of SQL Programming* by the authors of this [illegible]. These two books are intended as companions and together would form a basis for a [illegible] relational databases.

Personal Oracle 8, the PC version of the ORACLE relational database management system. It is important to note that many RDBMS vendors provide more interactive ways to create and maintain databases using tables and/or windows. The examples we will use in this area come from Microsoft Access. Although we will show alternate ways to maintain tables and create queries, the reader should remember that SQL is still the background language used by MS Access. From the MS Access query interface, one can always view the SQL that accomplishes the same purpose.

One of the main characteristics of the SQL language is that it is a *declarative* or *nonprocedural language*. From the programmer's point of view, this implies that the programmer does not need to specify step by step all the operations that the computer needs to carry out to obtain a particular result. Instead, the programmer indicates to the database management system what needs to be accomplished and then lets the system decide on its own how to obtain the desired result.

The statements or commands that comprise the SQL language are generally divided into two major categories or *data sublanguages*, DDL and DML, which were explained in Chapter 1. Each sublanguage is concerned with a particular aspect of the language. DDL includes statements that support the definition or creation of database objects such as tables, indexes, sequences and views. Some of the most commonly used DDL statements are the different forms of the `CREATE`, `ALTER` and `DROP` commands. DML includes statements that allow the processing or manipulation of database objects. Some of the most commonly used DML statements are the different modalities of the `SELECT`, `INSERT`, `DELETE` and `UPDATE` statements. It is important to observe that all objects created in a database are stored in the data dictionary or catalog.

The SQL language can be used interactively or in its embedded form. *Interactive SQL* allows the user to issue commands[2] directly to the DBMS and receive the results back as soon as they are produced. When *embedded SQL* is used, the SQL statements are included as part of a program written in a general-purpose language such as C, C++ or COBOL. In this case, we refer to the general-purpose programming language as the *host language*. The main reason for using embedded SQL is to use additional programming language features that are not generally supported by SQL.

When embedded SQL is used, the user does not observe directly the output of the different SQL statements. Instead, the results are passed back in variables or procedure parameters.

As a general rule, any SQL instruction that can be used interactively can also be used as part of an application program. However, the user needs to keep in mind that there may be some syntactical differences in the SQL statements when they are used interactively or when they are embedded into a program. In this chapter, we only consider SQL in its interactive form.

3.1.1 NAME CONVENTIONS FOR DATABASE OBJECTS

Database objects, including tables and attribute names, must obey certain rules or conventions that may vary from one RDBMS to another. Failing to follow the naming conventions of a particular RDBMS may cause errors. However, users are generally "safe" if they stay within the following guidelines.

- Names can be from 1 to 30 characters long (64 in MS Access) with the exception that the database's names may be limited to 8 characters as is the case with any ORACLE database.
- Names must begin with a letter (lower or upper-case); the remaining characters can be any combination of upper or lower-case letters, digits or the underscore character. MS Access allows spaces in the name, but then requires the use of square brackets, e.g. [My pets].

Example 3.1

Tell whether these names are usually valid or invalid according to the SQL naming conventions. If invalid, explain the reason.

a. Database names: *Employee, PurchaseOrders, Sales Data*
b. Table names: *PARTS, Employee_Birthdays, Vendor-Names*
c. Attribute names: *Computer_ID_Number, $_Amount_Paid, addresscitystatezipcode*

a. Database names: *Employee* — valid; *PurchaseOrders* — invalid because in many versions of SQL, including Oracle, database names are limited to 8 characters; *Sales Data* — invalid because spaces are not allowed except in MS Access along with square brackets ([Sales Data]).
b. Table names: *PARTS* — valid because the case does not matter — this would be the same as *parts*; *Employee_Birthdays* — valid because table names can be any length up to 30 and may contain the underline character; *Vendor-Names* — invalid because only letters, digits, and the underline character may be used.
c. Attribute names: *Computer_ID_Number* — valid; *$_Amount_Paid* — invalid because characters other than letters, digits, and the underscore character are not allowed; *addresscitystatezipcode* — valid but not very smart. Not only is it hard to read with no underscore characters between the words, but also it implies that many different items would be included in the one column. Choose the columns so that one item of data is stored for each record. This attribute should really be divided into four different columns.

3.1.2 STRUCTURE OF SQL STATEMENTS/SQL WRITING

is shown in Fig. 3-1. At this moment, the reader should not be concerned with the inner working of this statement but only with its structural aspects.

```
SELECT column-name-1, column-name-2,...column-name-N
FROM table-name
WHERE Boolean-condition
ORDER BY column-name [ASC | DESC][, column-name [ASC | DESC]...];
```

Fig. 3-1. Keywords and clauses in the structure of a SQL statement.

In this SQL statement, we can distinguish four keywords and four clauses. The keywords are shown in bold in Fig. 3-1. As indicated before, a keyword is a word that has a specific meaning within the language. To use a keyword other than in its specific context will generate errors. The four clauses of this SQL statement are underlined in Fig. 3-1. Notice that each clause starts with a keyword.

In the preceding statement, the first two clauses (SELECT and FROM) are mandatory and the last two (WHERE and ORDER BY) are optional. When describing the syntax of SQL statements we will indicate optional keywords or clauses by enclosing them in square brackets. Using this convention, we can rewrite the preceding statement as shown in Fig. 3-2.

```
SELECT column-name-1, column-name-2,...column-name-N
FROM table-name
[WHERE condition]
[ORDER BY column-name [ASC | DESC][, column-name [ASC | DESC]....]];
```

Fig. 3-2. Mandatory and optional clauses explicitly indicated in a SQL statement.

Notice that in the ORDER BY clause we have enclosed in square brackets the words ASC (ascending) and DESC (descending) separated by a '|' character. This character, sometimes called the "pipe" character, is used to separate the different options that a user can choose when writing a SQL statement. The user can choose one and only one from each set of options. Notice also that we have underlined the word ASC. This indicates that this word is a *default value*. That is, a value that will be used by the system when the user does not choose a different option from the set of available choices. In Fig. 3-2, whenever the ORDER BY clause is used and the user does not choose the DESC option the RDBMS will use the ASC option by default.

When writing SQL statements or commands it is useful to follow certain rules and guidelines to improve the readability of the statements and to facilitate their editing if this is necessary. Some of the guidelines that the reader should keep in mind are:

- SQL statements are not case sensitive. However, keywords that start a clause are generally written in upper case to improve readability of the SQL statements.
- SQL statements can be written in one or more lines. It is customary to write each clause in its own line.
- Keywords cannot be split across lines and, with very few exceptions, cannot be abbreviated.
- SQL statements end in a semicolon. This semicolon must follow the last clause of the statement but it does not have to be in the same line.

3.2 Table Creation

In any RDBMS, tables are the basic unit of data storage. Tables hold all of the user-accessible data. To create a table it is necessary to name the table and all the attributes that comprise it. In addition, for every attribute the user needs to define its data type and, if necessary, the appropriate constraint or constraints. The name of the table identifies it as a unique object within the RDBMS.[3] Column or attribute names serve to differentiate the attributes from one another. Attribute names must be unique within the table. The data type of each attribute defines the characteristics of its underlying domain. The constraint or constraints that may be defined for a column impose conditions that need to be satisfied by all the values stored in the column. Tables in SQL are created using the **CREATE TABLE** statement or command. Fig. 3-3 shows the basic form of this command. In this section, we will assume that every time a table is created there is no other table by the same name previously created by the same user in his or her *schema*,[4] as explained in Chapter 1. (If a table already exists with that name, and the table is to be redefined, use the **DROP TABLE** command.) In the database lingo, tables created by a user are said to be "owned" by the user.

```
CREATE TABLE table-name
(
  column-name-1        data-type-1    [constraint],
  column-name-2        data-type-2    [constraint],
       ..
  column-name-N        data-type-N    [constraint]
);
```

Fig. 3-3. A basic syntax of the CREATE TABLE command.

[3] Formally, it defines the table as a unique object within the user's tablespace or schema or within the entire system depending upon whether or not the table has been defined as public.

[4] This term refers to a collection of logical structures of data or schema objects. Each user owns a single schema whose name is that of its owner. Any user, with the appropriate privileges, can create objects in his or her own schema. Some schema objects are tables, synonyms, indexes, sequences and views.

When describing the syntax of the CREATE TABLE command we will call a *column definition line* every line of the form

column-name data type [constraint],

where optional elements are enclosed in square brackets. Therefore, according to this notation, every column definition line requires a column name and a data type. Constraints are optional. Usually, when typing a CREATE TABLE command, each column definition line is written in a separate line for readability. Commas separate column definition lines except for the last line which is followed by a parenthesis. As any other SQL command, a semicolon follows the closing parenthesis.

Example 3.2
Write the SQL statements to create a table called SAMPLE TABLE with two attributes: Attribute 1 is text no more than 15 characters long and Attribute 2 is currency. NOTE: Recall from Table 2-1 that in the Oracle implementation of SQL, text of varying length is indicated by `Varchar2(max-size)` and currency would simply be a number with two decimal places.

```
CREATE TABLE Sample_Table
(
 Attribute_1    Varchar2(15),
 Attribute_2    Number(4,2)
);
```

Notice in this code that the naming rules from Section 3.1.1 have been followed. Spaces in names of tables or attributes are not allowed and usually replaced with the underscore character.

3.2.1 CONSTRAINT IMPLICATIONS

The SQL standard requires that, whenever a constraint is defined, the constraint be given a name. Constraints can be named explicitly by the user at the time a table is created or modified. Otherwise, the constraint is named internally by the RDBMS. Constraints that are named by the user are called *named constraints*. Constraints named by the RDBMS are vendor dependent and are called *unnamed constraints*. Although constraint names can follow the conventions indicated in Section 3.1.1, we will use the following format for *named* constraints:

CONSTRAINT table-name-column-name-suffix

where the clause CONSTRAINT is *mandatory* and the suffix is a one or two letters sequence that indicates the type of the constraint. Table 3-1 shows a list of the suffixes that we will use in this book. Unnamed constraints *must not* be preceded by the CONSTRAINT clause.

Table 3-1. Suffix conventions for named constraints.

SUFFIX	MEANING
PK	Primary key
FK	Foreign key
NN	Not NULL
U	Unique

Some of the constraints that we will consider in this chapter are shown in Table 3-2. A constraint defined as part of a column definition is called a *column constraint*. This type of constraint can be used to impose a single condition on the column in which it is defined. A constraint that is part of a table definition is called a *table constraint*. This type of constraint can be used to define more than one constraint on any column of the table. We will only consider column constraints here. See a SQL reference for an explanation of table constraints.

Table 3-2. Basic column and table constraints.

CONSTRAINT	DEFINITION
NOT NULL (*)	Prevents NULL values from being entered into a column.
UNIQUE (**)	Prevents duplicate values from being entered into a column.
PRIMARY KEY (***)	Requires that all values entered into the column be unique and different than NULL.

* column_constraint.

** column_constraint or table constraint depending upon whether or not it is applied to one or more columns respectively.

*** column_constraint when defining a single PK and a table_constraint when defining a composite PK.

Example 3.3

A university allows students to buy meals using a Flex Card. They purchase a certain amount and each time they use the card the appropriate amount is subtracted from their account. Create the table `Flex_Card` with the attributes and assumptions indicated below. Choose the most appropriate data types.

Attributes: student name, card number, starting value, value left, and pin number.

Assumptions: The attribute student name may have up to 25 characters. The attributes value left and original value are measured in dollars and cents. The attribute card number may have up to 15 digits. The pin number attribute is always 12 char

```
CREATE TABLE Flex_Card
( Student_Name          VARCHAR2(25),
  Card_Number           VARCHAR2(15),
  Starting_Value        NUMBER(4,2),
  Value_Left            NUMBER(4,2),
  Pin_Number            CHAR(12)
);
```

The attribute Student_Name is obviously of type character. Since not all student names are 25 characters long, the data type of this column is Varchar2(25). The Card_Number and Pin_Number columns are both of character type because they are not involved in any type of computation. Since Card_Number may vary in length, its data type is Varchar2(15). The data type of Pin_Number is Char(12) since this column has a fixed length. The Starting_Value and Value_Left columns are both numerical quantities that may have up to 4 digits including 2 decimal places. Observe the use of the underscore character to improve the readability of the attribute names.

Example 3.4
Rewrite the CREATE TABLE of the previous example with the attribute Card_Number defined as the primary key and the attribute Pin_Number defined as unique. Use unnamed constraints.

In this case, we may use a column constraint to define the attribute Card_Number as the PK. By definition of PK, this attribute is also UNIQUE and therefore it is not necessary to define it as such. The attribute Pin_Number is only defined as a UNIQUE attribute. The reader should be aware that these two constraints behave a little bit differently. The PRIMARY KEY constraint, in addition to requiring that the values be unique, also guarantees that the values of the Card_Number column cannot be NULL. The UNIQUE constraint of the Pin_Number column does not allow duplicate values in this column but it does allow NULL values. The new CREATE TABLE command is shown below.

```
CREATE TABLE Flex_Card
( Student_Name          VARCHAR2(25),
  Card_Number           VARCHAR2(15) PRIMARY KEY,
  Starting_Value        NUMBER(4,2),
  Value_Left            NUMBER(4,2),
  Pin_Number            CHAR(12)UNIQUE
);
```

Example 3.5
Rewrite the CREATE TABLE of the previous example using named constraints.

Following the convention for naming constraints and using the suffixes of Table 3-1, the constraints associated with the attributes Card_Number and Pin_Number are respectively

Calling_Card_Card_Number_PK and Calling_Card_Pin_Number_U

The corresponding CREATE TABLE command is shown below:

```
CREATE TABLE Flex_Card
(Student_Name       VARCHAR2(25),
 Card_Number        VARCHAR2(15) CONSTRAINT
                    calling_card_card_number_PK PRIMARY KEY,
 Starting_Value     NUMBER(4,2),
 Value_Left         NUMBER(4,2),
 Pin_Number         CHAR(12) CONSTRAINT
                    calling_card_pin_number_U UNIQUE);
```

Notice that the column definition for the attributes `Card_Number` and `Pin_Number` have been written in more than one line to fit the width of this page.

3.2.2 CREATING TABLES AND CONSTRAINTS IN MS ACCESS

MS Access provides a way to create the table directly through the use of the table design view. Fig. 3-4 shows the window where the user can type in the field names, choose a data type, and give a brief description of the column. The table from Example 3.2 has been defined. Notice that table and attribute names are allowed to contain spaces. In Fig. 3-4, *Attribute 1* is the primary key, as indicated by the picture of the key to the left of the field name. Limits on size, default values, and many other constraints can be indicated through the use of the bottom half of the window. The *Indexed Yes(No duplicates)* will enforce the SQL Unique constraint, and if the line *Required* is set to *Yes* then NULL values are not allowed. Consult a reference on MS Access for a detailed guide to using other sections of this screen.

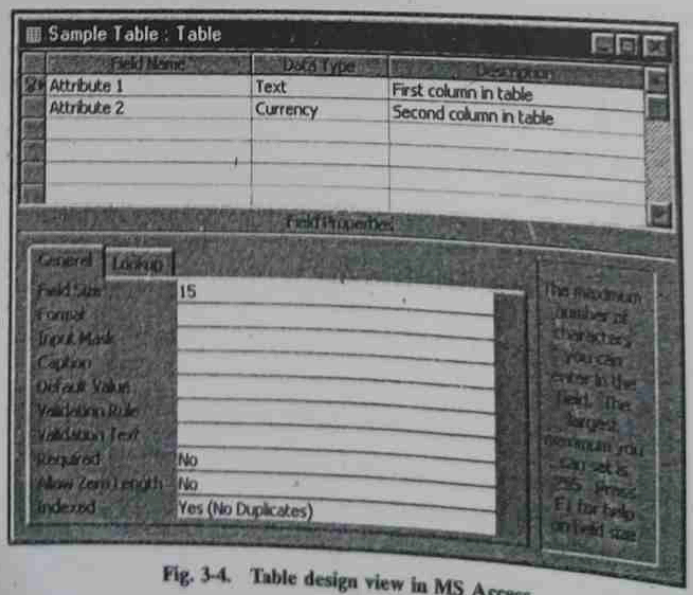
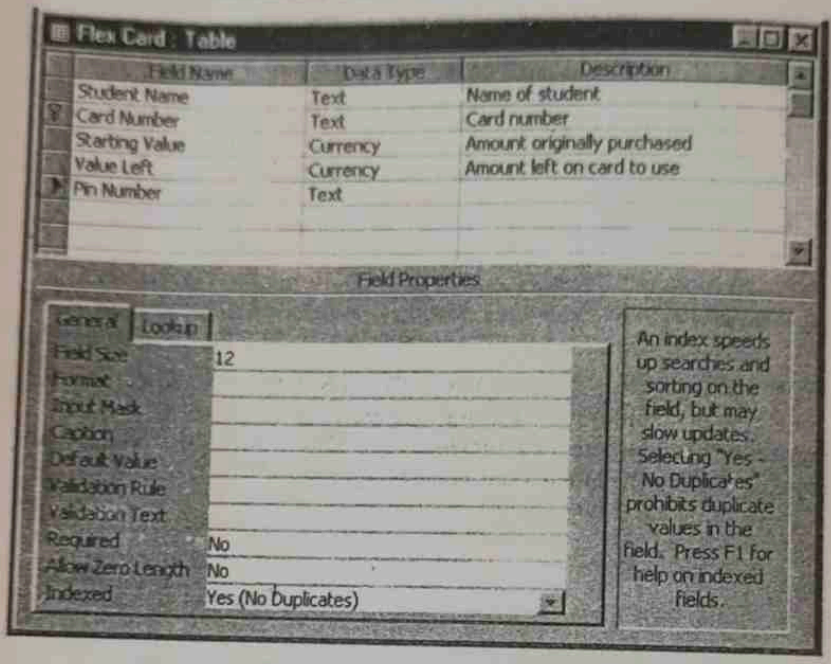


Fig. 3-4. Table design view in MS Access.

Example 3.6
Show the Flex Card table design as it would be created using MS Access. Be sure to use all the constraints shown in Example 3.4.



The design view of the table shows that the Card Number is the primary key. The Pin Number can be forced to be Unique by selecting the *No Duplicates* value for the property *Indexed*.

3.2.3 POPULATING AND MAINTAINING TABLES

After creating a table, the user may add rows to the table using the `INSERT INTO` command. The process of adding rows to a table is called *populating the table*. In its simplest form, this command allows the user to add rows to a table *one row* at a time. Fig. 3-5 shows the basic syntax of this command. More complex insertion techniques are available, but beyond the scope of this chapter. Section 2.7.1 described the care that must be taken when trying to insert a tuple into a table. The `INSERT INTO` commands must be formatted correctly and not violate any of the rules listed in Chapter 2. If the `INSERT INTO` operation fails, an error message will appear.

```
INSERT INTO table-name (column-1,.column-2,...column-N)
VALUES (value-1, value-2,...value-N);
```

Fig. 3-5. Basic form of the `INSERT` statement.

In Fig. 3-5, `column-1, column-2,...column-N` are the table's columns, and `value-1, value-2, value-3,...value-N` are the values that will be inserted into their corresponding columns. Notice that the value to be inserted into a column must be of the same data type that was specified for that column when its table was created. It is important to keep in mind that we *must* specify a value in the `VALUES` clause for each column that appears in the column list.

Example 3.7
Insert into the Flex_Card table the data indicated below.

Student_Name	Card_Number	Starting_Value	Value_Left	Pin Number
Ann Stephens	1237096435	20.00	12.45	987234569871
John Gilmore	5497443544	15.00	11.37	433809835833

Since the basic form of the `INSERT INTO` command only allows the insertion of one row at a time, it is necessary to use two consecutive `INSERT INTO` commands to add these two tuples to the Flex_Card table. Notice that all character data has been enclosed in single quotes.

```
INSERT INTO Flex_Card (Student_Name, Card_Number,
                       Starting_Value,Value_Left,Pin_Number)
VALUES ('ANN STEPHENS', '1237096435', 20.00, 12.45,
       '987234569871');
INSERT INTO Flex_Card (Student_Name, Card_Number,
                       Starting_Value,Value_Left,Pin_Number)
VALUES ('JOHN GILMORE', '5497443544', 15.00, 11.37,
       '433809835833');
COMMIT;
```

The `COMMIT` command that follows the last `INSERT INTO` command is necessary to make the changes to the table permanently. The `COMMIT` command will be explained in Chapter 6.

The reader should be aware that the order of the columns following the `INTO` clause is immaterial provided that their corresponding values appear in the same order in the `VALUES` clause. This allows us to fill in the columns of a row in any order. Example 3.8 illustrates this.

EXAMPLE 3.8
Insert into the Flex_Card table the rows shown below, and fill in the columns in the following sequence: `Pin_Number`, `Card_Number`, `Student_Name`, `Starting_Value`, and `Value_Left`.

Student_Name	Card_Number	Starting_Value	Value_Left	Pin Number
John Darc	2137096435	20.00	20.00	125234569871
Richard Lion	3817443544	20.00	20.00	632809835833

As in the previous example, we need two consecutive INSERT statements to add these rows to the Flex_Card table. The new INSERT statements are as follows:

```
INSERT INTO Flex_Card (Pin_Number,Card_Number;Student_Name,
                       Starting_Value,Value_Left)'
VALUES ('125234569871', '2137096435', 'JOAN DARC', 20.00,
        20.00);
INSERT INTO Flex_Card (Pin_Number, Card_Number, Student_Name,
                       Starting_Value,Value_Left)
VALUES ('632809835833', '3817443544', 'RICHARD LION',
        20.00, 20.00);
COMMIT;
```

As part of the normal maintenance of a database, one or more rows may need to be updated or removed from the database. For instance, in the Flex_Card table, the amount left on the card may change, or the student might leave school. The SQL command that allows the user to delete rows from a table is the DELETE command (see Section 2.7.2). The SQL command that allows the user to update values in existing rows is the UPDATE command (see Section 2.7.3).

The DELETE command can be used to remove rows that meet certain conditions or it can be used to remove all rows from a particular table. The syntax of this command to remove rows that meet certain conditions is shown here:

```
DELETE FROM table-name WHERE condition;
```

The syntax of the DELETE command to remove all rows of a table is

```
DELETE table-name;
```

Example 3.9

Student Joan Darc has left school. Remove her information from the database.

```
DELETE FROM Flex_Card
WHERE student_name = 'JOAN DARC';
```

This DELETE command will remove the entire row from the table. Sometimes, however, information in a row must be changed. As the name of the UPDATE command suggests, its primary function is to update the rows of a table. The basic syntax of this SQL command to update one or more values of a single row is as follows:

```
UPDATE table-name
SET col-1 = new-val1 [..., ...col-N = new-valN]
[WHERE condition];
```

where col-1..., col-N stand for column names and new-val1,...new-valN stand for the new values that will be stored in their corresponding columns. The WHERE clause allows us to change the values of selected rows. The following example illustrates the use of this command.

Example 3.10
Richard Lion has spent $7.50 of the value of his flex card. The new amount left on his card is $13.50. Update his row.

```
UPDATE Flex_Card
SET value_left = 13.50
WHERE student_name = 'RICHARD LION';
```

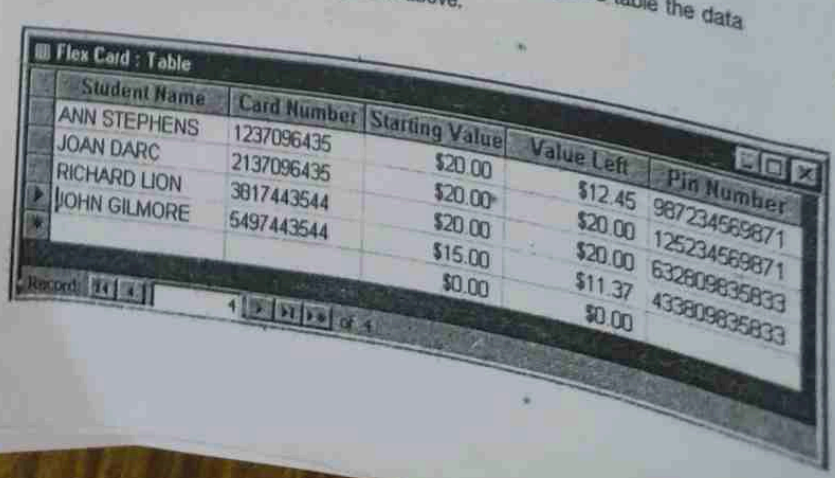
The UPDATE statement sets the specified attribute to the new value. Several columns and rows can be updated at the same time through more complex UPDATE commands.

3.2.4 POPULATING TABLES IN MS ACCESS

MS Access provides a way to enter items into the table from the table view without the use of SQL. Rows can be inserted, deleted, or individual values can be updated directly. Fig. 3-6 shows this view of the table. Items can simply be updated or typed into each column. During the editing process, constraints are enforced as the user is typing in the information. The editor flags the error and does not allow the row to be recorded until unique or required items are correct.

Fig. 3-6. Table view in MS Access.

Example 3.11
Show the screen that would be used to type into the Flex_Card table the data from both Example 3.7 and Example 3.8 above.



After the data is typed directly into the table, when the user closes the screen MS Access asks if the changes to the table should be saved. However, the user should be aware of the need to save frequently as data are entered. Saving is the equivalent of the COMMIT in Oracle.

3.3 Selections, Projections, and Joins Using SQL

The **SELECT** statement is the most frequently used SQL statement. The SELECT statement is used primarily to *query* or retrieve data from the database. For this reason, it is customary to call a SELECT statement a *query*. In this book, we will refer to any SELECT statement by this name. The basic syntax of this statement is shown once again in Fig. 3-7.

```
SELECT column-1, column-2, column-3,...,column-N
FROM table-1,..., table-N
[WHERE condition]
[ORDER BY column-name [ASC|DESC][,column-name [ASC|DESC]...]];
```

Fig. 3-7. Basic structure of the SELECT statement.

The SELECT statement is formed by at least two *clauses*: the **SELECT** clause and the **FROM** clause. The clauses **WHERE** and **ORDER BY** are optional. Observe that the SELECT statement, like any other SQL statement, ends in a semicolon. The functions of each of these clauses are summarized below.

- The **SELECT** clause lists the subset of attributes or columns to retrieve. (See Projection in Section 2.5.2.) The attributes listed in this clause are the columns of the resulting relation.
- The **FROM** clause lists the tables from which to obtain the data. The columns mentioned in the SELECT clause must be columns of the tables listed in the FROM clause. (To accomplish Equijoins, as explained in Section 2.5.3, several tables may be used.)
- The **WHERE** clause specifies the condition or conditions that need to be satisfied by the tuples or rows of the tables indicated in the FROM clause. (See Selection in Section 2.5.1.)
- The **ORDER BY** clause indicates the criterion or criteria used to sort rows that satisfy the WHERE clause. The ORDER BY clause only affects the display of the data retrieved, not the internal ordering of the rows within the tables.

As a mnemonic aid to the basic structure of the SELECT statement some authors summarize its functionality by saying that "you **SELECT** columns **FROM** tables **WHERE** the rows satisfy certain conditions and the result is **ORDERED BY** specific columns".

you might want to print out all the names of people in a certain zip code area or you might want to see the names of all employees that work for a given department. The condition that accompanies the where clause defines the criterion to be met. The types of condition that we consider in this chapter are of the following form:

```
Column-name    comparison-operator    single-value
```

Column-name is the name of one of the columns of the table indicated in the FROM clause. The comparison operator is one of the operators shown in Table 3-5. By a single value we mean a numeric quantity or a character string. It is possible to construct other more complex queries using compound conditions using the Boolean operators AND, OR and NOT.

Table 3-5. Comparison operators for the WHERE clause.

Comparison Operator	Description
=	equal to
<>	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

The following example illustrates the use of the SELECT command.

Example 3.12
Recall the EMPLOYEE table from Example 2.8.

EMPLOYEE

ID	NAME	DEPT	TITLE
100	Smith	Sales	Clerk
200	Jones	Marketing	Clerk
300	Martin	Accounting	Clerk
400	Bell	Accounting	Sr. Accountant

Using the EMPLOYEE table, display the name and title of all the employees who work in the accounting department.

To retrieve this data from the EMPLOYEE table we use the following statement

```
SELECT name, title, dept      <-- Columns to retrieve
FROM employee                 <-- Table from which to retrieve the data
WHERE dept = 'Accounting';    <-- Criterion to be satisfied
```

The resulting table is shown below.

NAME	TITLE	DEPT
Martin	Clerk	Accounting
Bell	Sr. Accountant	Accounting

Notice that in the resulting table the attribute names and their corresponding values are displayed in the same order in which they were listed in the SELECT statement. All the retrieved tuples satisfy the condition indicated in the WHERE clause. That is, for any tuple t of the resulting relation, t(DEPT) = Accounting. Observe that the rows retrieved are not sorted by alphabetical name. However, if we want to display the table with the names ordered in alphabetical order, we can proceed as indicated in Example 3.13. Observe also that in the SELECT clause the attribute names are not case sensitive. That is, we can write the attributes in either lower or upper case and obtain the same result. However, the condition indicated in the WHERE clause requires some consideration. Since DEPT is a character column, the condition has been enclosed in single quotes. In addition, we need to remember that character data is case sensitive. Had the condition of the WHERE clause been written as DEPT='ACCOUNTING' then no tuple would have satisfied it. Observe that the strings 'Accounting' and 'ACCOUNTING' are different strings.

Example 3.13
Display the result of the previous query in alphabetical order by the employee's name.

In this case, the SELECT statement needs to indicate to the RDBMS that the results need to be sorted in alphabetical order. Since we want to sort the resulting table according to the attribute NAME, it is necessary to mention this attribute in the ORDER BY clause. By default, the sorting of rows is done in ascending order[5] according to the column or columns that define the order, in this case, the attribute NAME. The SELECT statement to accomplish the desired result is shown next.

[5] By ascending order, we mean from lower values to higher values according to the coalescence sequence of the ASCII characters.

```
SELECT name, title, dept
FROM employee
WHERE dept = 'Accounting'
ORDER BY name;
```

The resulting table is shown here:

NAME	TITLE	DEPT
Bell	Sr. Accountant	Accounting
Martin	Clerk	Accounting

The rows of this table have been sorted in alphabetical order by last name.

The SELECT statement allows the use of the asterisk as a wildcard character. The use of SELECT * is the same as asking for all the columns of the table to be shown in the resulting chart.

Example 3.14
Display all the information for everyone in the EMPLOYEE table who is a clerk.

```
SELECT *
FROM employee
WHERE title = 'Clerk';
```

ID	NAME	DEPT	TITLE
100	Smith	Sales	Clerk
200	Jones	Marketing	Clerk
300	Martin	Accounting	Clerk

Once again the resulting table would not be displayed alphabetically unless we used the ORDER BY clause.

The SELECT statement can accomplish the Equijoin operation as explained in Chapter 2 by accessing columns from two or more tables. As stated in Section 2.5.3, a join consists of all the tuples resulting from concatenating the tuples of the first relation with the tuples of the second relation that have identical values for a common set of attributes. Recall the second table, DEPARTMENT, from Example 2.8.

DEPARTMENT

ID	DEPT	LOCATION
100	Accounting	Miami
200	Marketing	New York
300	Sales	Miami

Example 3.15
Display all the information about the employees along with their department's ID, name and location. For this, you will need two tables, DEPARTMENT and EMPLOYEE.

```
SELECT department.id, department.dept, department.location,
       employee.id, employee.name, employee.title
FROM department, employee
WHERE department.dept = employee.dept;
```

This query requires the join of two separate tables, as evident in the FROM clause. The resulting chart from this query would produce the same table below as shown in Example 2.8.

DEPT ID	DEPARTMENT NAME	LOCATION	EMPLOYEE ID	EMPLOYEE NAME	TITLE
100	Accounting	Miami	300	Martin	Clerk
100	Accounting	Miami	400	Bell	Sr. Accountant
200	Marketing	New York	200	Jones	Clerk
300	Sales	Miami	100	Smith	Clerk

Since for a true join we only wanted the row for each employee matched with his or her department, we needed to use the WHERE clause to connect columns in each table. If the WHERE clause had been omitted, the resulting chart would have contained twelve rows, one for each row in DEPARTMENT matched with each row in EMPLOYEE. This resulting table, shown below, is the Cartesian product of the two relations. This Cartesian product has 6 attributes, 3 (from EMPLOYEE) + 3 (from DEPARTMENT) = 6, and 12 tuples, 4 (from EMPLOYEE) * 3 (from DEPARTMENT) = 12.

ID	DEPT	LOCATION	ID	NAME	TITLE
100	Accounting	Miami	100	Smith	Clerk
200	Marketing	New York	100	Smith	Clerk
300	Sales	Miami	100	Smith	Clerk
100	Accounting	Miami	200	Jones	Clerk
200	Marketing	New York	200	Jones	Clerk
300	Sales	Miami	200	Jones	Clerk
100	Accounting	Miami	300	Martin	Clerk
200	Marketing	New York	300	Martin	Clerk
300	Sales	Miami	300	Martin	Clerk
100	Accounting	Miami	400	Bell	Sr. Accountant
200	Marketing	New York	400	Bell	Sr. Accountant
300	Sales	Miami	400	Bell	Sr. Accountant

Remember to use caution when finding the Cartesian product of two tables because often some of the tuples make no sense. For example, the first tuple shows that Smith is a Clerk in the Accounting department, 100, which is located in Miami. The second tuple, however, says that Smith is a Clerk in Department 100, and the Marketing Department, 200, is in New York. Ordinarily, one would have no use for that information.

3.3.1 SET OPERATIONS IN SQL

Section 2.6 explained the mathematical basis of set operations performed on relations. This section will demonstrate the use of SQL to perform these operations. The first example is a union of two relations, or the set of all tuples in both relations with no duplicate tuples listed. The UNION operator is shown in Fig. 3-8. Recall from the definition of union in Chapter 2 that the tables must have the same structures.

```
SELECT *
FROM table_1
UNION
SELECT *
FROM table_2;
```

Fig. 3-8. UNION operator in SQL.

Example 3.16
Recall these tables from Chapter 2. Write the SQL code that will perform the UNION of the two tables, or show all the programmers in the organization.

C_PROGRAMMER

Employee_Id	Last_Name	First_Name	Project	Department
101123456	Venable	Mark	E-commerce	Sales Department
103705430	Cordani	John	Firewall	Information Technology
101936822	Serrano	Areant	E-commerce	Sales Department

JAVA_PROGRAMMER

Employee_Id	Last_Name	First_Name	Project	Department
101799332	Barnes	James	Web Application	Information Technology
101936822	Serrano	Areant	E-commerce	Sales Department

```
SELECT *
FROM c_programmer
UNION
SELECT *
FROM java_programmer;
```

The resulting chart demonstrates the union of the two tables. All the tuples from each relation are included with no duplicate rows.

```
EMP_ID    LAST    FIRST  PROJECT         DEPARTMENT
--------- ------- ------ --------------- ----------------------
101123456 Venable Mark   E-commerce      Sales
101799332 Barnes  James  Web Application Information Technology
101936822 Serrano Areant E-commerce      Sales
103705430 Cordani John   Firewall        Information Technology
```

Intersection of relations can also be performed in SQL. The intersection is the set of tuples present in both relations. The SQL syntax is shown in Fig. 3-9.

```
SELECT *
FROM table_1
INTERSECT
SELECT *
FROM table_2;
```

Fig. 3-9. INTERSECTION operator in SQL.

Example 3.17
Write the SQL code that will perform the INTERSECTION of the two programmer tables, or show which programmers can program in both C and JAVA.

CHAPTER 4

Functional Dependencies

4.1 Introduction

In any relational database, controlling the redundancy and preserving the consistency of data are two of the most important issues that any database designer or data administrator has to face. *Data redundancy* occurs when a piece of data is stored in more than one place in the database. If the content of that piece of data is changed to a particular value, then it is necessary to ensure that every copy of the same piece of data is changed to the same value. This piece of data is said to be consistent in the database. If some, but not all, copies of this piece of data are changed to the same value, the data is said to be inconsistent. A database is said to be in a *consistent state* if all its data is consistent. Otherwise it is said to be in an *inconsistent state*. Since relations are the logical entities that store data in any RDBMS, to achieve the goal of controlling data redundancy and maintaining its correctness and accuracy it is necessary to be aware of all constraints that apply to the database relations.

One way to learn more about the different types of constraints imposed on all permissible data of a relation or set of relations is through the use of functional dependencies (FDs). *Functional dependencies* arise naturally in many ways due to requirements or restrictions that exist in the real world and that have to be captured by the database. In general, these restrictions or constraints can be classified into two general groups: semantic constraints and agreement constraints. Semantic constraints depend on the meaning or understanding of the attributes of a relation. For instance, in an EMPLOYEE relation no employee may have a negative salary or a negative age. Agreement or concordance

constraints do not depend on the particular values of the attributes of a tuple, but on whether or not tuples that agree on certain attributes agree also in the values of some of their other attributes. For instance, consider the attributes Department and Supervisor of an EMPLOYEE relation. If we assume that employees only work for one department, that there is only one supervisor per department and that every department has a supervisor then two tuples with the same value under the Supervisor column must have the same value under the Department column. Functional dependencies are the most important of the agreement or concordance constraints. We will consider this topic next.

4.2 Definition of Functional Dependencies

Given a relation r(R) and two sets of its attributes A and B. We will say that *attribute(s) A functionally determines attribute(s) B with respect to r*, denoted by $A \rightarrow B$, if and only if for any two tuples t_1 and t_2 of r whenever $t_1(A) = t_2(A)$ then $t_1(B) = t_2(B)$. That is, if A functionally determines B, then whenever two tuples of r have identical values in column A their respective values in column B must also be identical. If the relation r is understood, we will just say that A functionally determines B. In the notation $A \rightarrow B$, attribute A, the left-hand side of the functional dependency, is called the *determinant*; attribute B is called the right-hand side of the functional dependency. If $A \rightarrow B$ (with respect to a relation r), we will say the relation r "satisfies the functional dependency" or that the "functional dependency is satisfied by the relation". Observe that in the definition given above, both A and B have been defined as sets of attributes and not necessarily as single attributes. In other words, A and B may be composite attributes. When more than one attribute are explicitly indicated in a functional dependency, it is customary to use concatenation to stand for set union between sets of operators. That is, in the functional dependency $AB \rightarrow X$, AB is shorthand for $A \cup B$. In addition, the notation $A \not\rightarrow B$ is used to denote that a set of attributes A does not functionally determine a set of attributes B. Functional dependencies where the right-hand side consists of only one attribute are called *simple* FDs.

Example 4.1

Given the Lakes-Of-The-World relation shown below, state whether or not the functional dependencies (a) Continent → Name and (b) Name → Length are satisfied by this relation. Assume that the Area attribute is measured in square miles and that the Length attribute is measured in miles.[1]

Lakes-Of-The-World

Name	Continent	Area	Length
Caspian Sea	Asia-Europe	143244	760
Superior	North America	31700	350
Victoria	Africa	26828	250
Aral Sea	Asia	24904	280
Huron	North America	23000	206
Michigan	North America	22300	307
Tanganyika	Africa	12700	420

a. The functional dependency Continent → Name *is not* satisfied by the relation. We can verify this because according to the definition of functional dependency, if Continent → Name, then for any two tuples of the relation that have the same value under the Continent attribute their values under the Name attribute must also be the same. For example, in the Lakes-Of-The-World relation consider the following tuples:

t_1: (Superior, North America, 31700, 350) and t_2: (Huron, North America, 23000 206).

Notice that

t_1(Continent) = t_2(Continent) = North America but
t_1(Name) = Superior ≠ t_2(Name) = Huron.

Since these two tuples agree on their values under the Continent attribute but not on their values under the Name attribute, we have that Continent ↛ Name.

b. The functional dependency Name → Length is satisfied by the relation. In this case, the relation is obviously satisfied because for any given lake there is only one length associated with it.

The reader should keep in mind that determining whether or not an attribute functionally determines another attribute is based only on the meaning of the attributes. In this sense, functional dependencies can be considered assertions about the real world that should hold at any point in time. That is, they should be true for all instances of the relation. Since functional dependencies depend on the semantics of their attributes they should not be inferred from the current content of a relation. That is, we cannot look at the current content of a relation and based on the values of the attributes A and B decide that A → B. Notice also that from the definition of functional dependency (FD) and the previous discussion it is *always* possible to determine if a *given* functional dependency is

or is not satisfied by an instance of a relation. However, it is not valid to infer functional dependencies from a particular instance of a relation without first taking into account the meaning of the intervening attributes. The Satisfies algorithm shown below can be used to determine if a relation r satisfies or does not satisfy a given functional dependency $A \rightarrow B$. The input to the algorithm is a given relation r and a functional dependency $A \rightarrow B$. The output of the algorithm is True if r satisfies $A \rightarrow B$; otherwise the output is False.

The Satisfies Algorithm[2]

(1) Sort the tuples of the relation r on the A attribute(s) so that tuples with equal values under A are next to each other.

(2) Check that tuples with equal values under attribute(s) A also have equal values under attribute(s) B.

(3) If any two tuples of r meet condition 1 but fail to meet condition 2 the output of the algorithm is False. Otherwise, the relation satisfies the functional dependency and the output of the algorithm is True.

Example 4.2
Using the relation of the previous example, apply the Satisfies algorithm to show that the attribute Continent does not determine the attribute Name.

Lakes-Of-The-World

Name	Continent	Area	Length
Victoria	Africa	26828	250
Tanganyika	Africa	12700	420
Aral Sea	Asia	24904	280
Superior	North America	31700	350
Huron	North America	23000	206
Michigan	North America	22300	307

At least two of the values of this attribute are different from each other for identical values of the Continent attribute.

At least two of the values of this attribute are equal to each other.

[2] Adapted from *The Theory of Relational Databases* by D. Maier, Computer Science Press, 1983.

After sorting the tuples of the relation on the Continent attribute, notice that the use of the algorithm makes it obvious that the relation does not satisfy the functional dependency that is Continent → Name. The output of the algorithm is False.

4.3 Functional Dependencies and Keys

Given a relation r(R) and its primary key K, for any particular value of K we can always determine whether or not there is a tuple in the relation with that K value. In addition, if the tuple is present in the relation we can not only uniquely identify the tuple but also the value of any of its attributes. Using the notion of functional dependency we can say that the key K functionally determines any attribute of the relation. That is, $K \rightarrow A_i$ where A_i is any set of attribute(s) of the relation. Since keys are particular cases of FDs all properties that are true for FDs are also valid for keys. Some of these properties are considered in the next and following sections. As we indicated in Chapter 2, Section 2.3, we call prime attributes any set of attributes that comprises a PK or an alternate key.

4.4 Inference Axioms for Functional Dependencies

A relation r may satisfy more than one set of functional dependencies. In theory, it is possible to use the Satisfies algorithm with all possible combinations of attributes of the relation r to determine which FDs are satisfied by the relation. Although this method may indicate all FDs that are satisfied by the relation, from a practical point of view, it is obviously a tedious and time-consuming task. The *inference axioms* offer an alternate method that allows us to infer the FDs that are satisfied by a relation without the use of any algorithm. Given a set F of FDs, the inference axioms are a set of rules that tell us that if a relation satisfies the FDs of F the relation must satisfy certain other FDs. The latter set of FDs that the relation must satisfy are said to be *derived* or *logically deduced* from the FDs of F.

The inference axioms are said to be *complete* and *sound*. By complete we mean that, given a relation r(R) and a set F of functional dependencies satisfied by r, the axioms allow us to derive all valid functional dependencies that are satisfied by r. By sound we mean that if the axioms are correctly applied they cannot derive false dependencies.

Assume that r(R) is a relation and that X, Y, Z and W are subsets of R. The set of inference axioms[3] is shown below. We have enclosed in parentheses the most common name or names for each of the individual axioms. Whenever two or more names are listed we will generally use the first one in the list.

[3] This set of axioms is sometimes called Armstrong's Axioms after W. W. Armstrong who initially proposed a set of somewhat similar axioms.

Inference Axioms

(1) If $Y \subseteq X$, then $X \rightarrow Y$ (Reflexivity*)
(2) If $X \rightarrow Y$, then $XW \rightarrow Y$ and/or $XW \rightarrow YW$ (Augmentation*)
(3) If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$ (Transitivity)
(4) If $X \rightarrow Y$ and $YW \rightarrow Z$, then $XW \rightarrow Z$ (Pseudotransitivity*)
(5) If $X \rightarrow Z$ and $X \rightarrow Y$, then $X \rightarrow YZ$ (Additivity or Union)
(6) If $X \rightarrow YZ$, $X \rightarrow Y$ and $X \rightarrow Z$ (Projectivity or Decomposition)

* This axiom is considered a basic axiom. Any other inference rule can be derived from a set that contains all the basic axioms.

Notice that when there are more than one attribute in either the left or right or both sides of a functional dependency the order in which these attributes are written is immaterial since the union set operation is commutative. That is, $AB \rightarrow CD$ can be written as $AB \rightarrow DC$ or $BA \rightarrow DC$ or $BA \rightarrow CD$ without altering the meaning of the functional dependency.

The axiom of *Reflexivity* indicates that given a set of attributes the set itself functionally determines any of its own subsets. As a particular case of this axiom we have that $X \rightarrow X$ for any set of attributes X. This is an immediate consequence of $X \subset X$. Functional dependencies of the form $X \rightarrow Y$ where $Y \subset X$ are called *trivial dependencies*.

The axiom of *Augmentation* indicates that we can augment or enlarge the left side of an FD or both sides conveniently with one or more attributes. Notice that the axiom does not allow augmenting the right-hand side alone.

The axiom of *Transitivity* indicates that if one attribute uniquely determines a second attribute and this, in turn, uniquely determines a third one, then the first attribute determines the third one.

The axiom of *Pseudotransitivity* is a generalization of the Transitivity axiom (see Supplementary Problem 4.16). Notice that for this axiom to be applied it is required that the entire right-hand side of a FD appears as attribute(s) of the determinant of another FD.

The axiom of *Additivity* indicates that if there are two FDs with the *same* determinant it is possible to form a new FD that preserves the determinant and has as its right-hand side the union of the right-hand sides of the two FDs.

The axiom of *Projectivity* or *Decomposition* is the inverse of the additivity axiom. This axiom indicates that the determinant of any FD can uniquely determine any individual attribute or any combination of attributes of the right-hand side of the FD.

The reader should keep in mind that the inference axioms allow us to discover new FDs that are present or satisfied by a relation even if they are not explicitly stated. For example, if a relation r satisfies $X \rightarrow Y$ and $Y \rightarrow Z$ it also satisfies $X \rightarrow Z$ even if the latter FD is not explicitly indicated as being satisfied by the relation.

As indicated before, the set of inference axioms allows us to derive new functional dependencies from a given set of FDs. The next example illustrates the use of the inference axioms and how they are applied to derive FDs.

Example 4.3
Given the set F = {A → B, C → X, BX → Z} derive AC → Z using the inference axioms. Assume that all these attributes belong to a relational scheme R not shown here.

(1) With A → B (given) and BX → Z (given) and by application of the axiom of Pseudotransitivity we have that AX → Z. Notice that we were able to apply this axiom because the attribute B appears on the right-hand side of a functional dependency and on the left-hand side of another as required by the Pseudotransitivity axiom.

(2) Using AX → Z (from the previous step) and C → X (given) and by application of the Pseudotransitivity axiom we have that AC → Z.

Example 4.3 illustrates the notion of logical implication between functional dependencies. In fact, this result shows that if a relation r satisfies the functional dependencies of the set F given above then it must satisfy the functional dependency AC → Z. We can formalize this concept as follows: Given a set F of functional dependencies of a relational scheme R, and a functional dependency X → Y. We say that F *logically implies* X → Y, denoted by F ⊨ X → Y, if every relation r(R) that satisfies the dependencies in F also satisfies X → Y.

Example 4.4
Given F = {A → B, C → D} with C ⊂ B, show that F ⊨ A → D.

(1) Knowing that C ⊂ B and using the axiom of Reflexivity we have that B → C.

(2) Using A → B (given) and B → C (from the previous step) and by application of the Transitivity axiom we have that A → C.

(3) From A → C (from step 2) and C → D (given) and by application of the Transitivity axiom we have that A → D.

Since A → D can be derived from the set of FDs of F then F ⊨ A → D.

4.5 Redundant Functional Dependencies

Given a set F of FDs, a functional dependency A → B of F is said to be *redundant with respect to the FD* ... be derived from the set of ... of F not

F of FDs is understood, we will say that the functional dependency $A \rightarrow B$ is redundant. Eliminating redundant functional dependencies allows us to minimize the set of FDs. The concept of minimality will be further explored in Section 4.6.4. Determining which FDs are redundant in a given set can be a tedious and long process, particularly when there are a large number of FDs in the set. The Membership algorithm shown below provides us with a more systematic procedure to determine redundant FDs; however, the algorithm is still lengthy if applied to a large number of FDs. The input to this algorithm is a set F of FDs and a particular FD of F that is being tested for redundancy. This algorithm is shown next.

The Membership Algorithm

Assume that F is a set of functional dependencies with $A \rightarrow B \in F$. To determine if $A \rightarrow B$ is redundant with respect to the other FDs of the set F proceed as follows:

(1) Remove temporarily $A \rightarrow B$ from F and initialize the set of functional dependencies G to F. That is, set $G = F - \{A \rightarrow B\}$. If $G \neq \emptyset$ proceed to step 2; otherwise stop executing the algorithm since $A \rightarrow B$ is nonredundant.

(2) Initialize the set of attributes T_i (with $i = 1$) with the set of attribute(s) A (the determinant of the FD under consideration). That is, set $T_i = T_1 = \{A\}$. The set T_1 is the current T_i.

(3) In the set G, search for functional dependencies $X \rightarrow Y$ such that all the attributes of the determinant X are elements of the current set T_i. There are two possible outcomes to this search:

(3-a) If such functional dependency is found, add the attributes of Y(the right-hand side of the FD whose determinant is in T_i) to the set T_i and form a new set $T_{i+1} = T_i \cup Y$. The set T_{i+1} is now the current T_i.
Check if all the attributes of B (the right-hand side of the functional dependency under consideration) are members of T_{i+1}. If this is the case, stop executing the algorithm because the FD: $A \rightarrow B$ is redundant. If not all attributes of B are members of T_{i+1}, remove $X \rightarrow Y$ from G and repeat step 3.

(3-b) If $G = \emptyset$ or there are no FDs in G that have all the attributes of its determinant in the current T_i then $A \rightarrow B$ is not redundant.

As indicated before, if a functional dependency $A \rightarrow B \in F$ is found to be redundant, we can remove it permanently from the set F.
The following example illustrates the use of the Membership algorithm.

Example 4.5
Given the set $F = \{X \rightarrow YW, XW \rightarrow Z, Z \rightarrow Y, XY \rightarrow Z\}$, determine if the functional dependency $XY \rightarrow Z$ is redundant in F.

Step (1) Temporarily remove the functional dependency $XY \rightarrow Z$ from the set F.

$$\text{Set } G = \{X \rightarrow YW, XW \rightarrow Z, Z \rightarrow Y\}$$

Step (2) Initialize a set of attributes T_1 with the attribute(s) of the determinant of the FD under consideration. In this particular case, since XY is the determinant of $XY \rightarrow Z$, we have that $T_1 = \{XY\}$.

Step (3/3-a) In the set G, look for functional dependencies such that all the attributes of their determinants are elements of the set T_1. Notice that the determinant of $X \rightarrow YW$ is an element of T_1. That is, $X \in T_1$. Adding to T_1 the attributes that appear on the right-hand side of this FD we form the new set shown below.

$$T_2 = T_1 \cup \{YW\} = \{XY\} \cup \{YW\} = \{XYW\}$$

Since Z, the right-hand side of $XY \rightarrow Z$ is not an element of T_2 ($Z \notin T_2$) remove $X \rightarrow YW$ from G. Set $G = \{XW \rightarrow Z, Z \rightarrow Y\}$ and repeat step 3.

Step (3/3-a) (2nd time) Observe now that all the attributes of the determinant of $XW \rightarrow Z$ are elements of T_2. Adding the attribute of the right-hand side of this FD to the set T_2 produces a new set $T_3 = T_2 \cup \{Z\} = \{XYWZ\}$. Since $Z \in T_3$ the algorithm stops and $XY \rightarrow Z$ is a redundant FD. That is, $XY \rightarrow Z$ can be safely removed from F.

To verify that $XY \rightarrow Z$ is redundant, we need to exclude this FD from F before attempting to derive it from the FDs of F using the inference axioms.

$$F = \{X \rightarrow YW, XW \rightarrow Z, Z \rightarrow Y\}$$

a. Using $X \rightarrow YW$ (given) and by application of the Projectivity axiom we have that $X \rightarrow Y$ and $X \rightarrow W$.
b. With $X \rightarrow W$ (from step a) and $XW \rightarrow Z$ (given) and by application of the Pseudotransitivity axiom we obtain $XX \rightarrow Z$.
c. Since the concatenation of attributes is equivalent to their union we have that $X \cup X = X$. Using this result and rewriting the determinant of the functional dependency of step b, we obtain $X \rightarrow Z$.

Therefore, the functional dependency $XY \rightarrow Z$ is redundant because we have shown that it can be derived from the FDs of F.

4.6 Closures, Cover and Equivalence of Functional Dependencies

Given a set F of FDs we are interested in determining all the FDs that can be logically implied by F. The set of all FDs that can be logically implied from F sees its most important application in the normalization process of relations (see Chapter 5). The following subsections provide the definitions and algorithms to generate a set of FDs or to test if a given set F of FDs implies a particular FD.

4.6.1 CLOSURE OF A SET F OF FUNCTIONAL DEPENDENCIES

Given a set F of functional dependencies for a relation scheme R, we define F^+, the *closure of F*, to be the set of all functional dependencies that are logically implied by F. In mathematical terms, $F^+ = \{X \rightarrow Y / F \models X \rightarrow Y\}$. The closure set satisfies the two following properties simultaneously:

(1) F^+ is the smallest set that contains F and satisfies property 2.

(2) Any application of the inference axioms to the FDs of F only produces FDs that are already in F^+

The following example illustrates this concept.

Example 4.6
Given set $F = \{XY \rightarrow Z\}$ determine all the elements of F^+. Assume that the scheme is comprised by all attributes mentioned in the FDs of F.

To generate all FDs that can be derived from F we proceed as follows: First, apply the inference axioms to all single attributes. Second, apply the inference axioms to all combinations of two attributes and use the functional dependencies of F whenever it is applicable. Next apply the inference axioms to all combinations of three attributes and use the FDs of F when necessary. Proceed in this fashion for as many different attributes as there are in F. The resulting set is shown below.

$F^+ = \{X \rightarrow X, Y \rightarrow Y, Z \rightarrow Z, XY \rightarrow X, XY \rightarrow Y, XY \rightarrow XY, XZ \rightarrow X, XZ \rightarrow Z,$
$XZ \rightarrow XZ, YZ \rightarrow Y, YZ \rightarrow Z, YZ \rightarrow YZ, XYZ \rightarrow XY, XYZ \rightarrow XZ,$
$XYZ \rightarrow YZ, XYZ \rightarrow XYZ\}$

CHAPTER 5

The Normalization Process

5.1 Introduction

In relational databases the term *normalization*[1] refers to a reversible step-by-step process in which a given set of relations is replaced by successive collections of relations that have a progressively simpler and more regular structure. Each step, referred to as a *normal form*, defines a set of criteria (the normal form test) that needs to be met by the different tables of the database. In this sense, to say that a relation is in a particular normal form is an indication of the conditions that the table has met. Since the process is reversible, the original set of relations can be recovered with no loss of information. As the normalization progresses to higher forms, the individual collection of relations becomes progressively more restricted on the type of functional dependencies that they can satisfy and the data anomalies that they can experience. Data anomalies will be explained in Section 5.3.

The objectives of the normalization process are:[2]

- To make it feasible to represent any relation in the database.
- To obtain powerful relational retrieval algorithms based on a collection of primitive relational operators.
- To free relations from undesirable insertion, update, and deletion anomalies.
- To reduce the need for restructuring the relations as new data types are introduced.

[1] This process was initially defined by E. F. Codd in "Normalized Data Base Structure: A Brief Tutorial," *Proc. ACM SIGFIDET Workshop on Data Description, Access and Control* pp. 1–17, 1971.

[2] Adapted from *Database Management Systems* by D. Tsichritzis and F. Lochovsky, Academic Press, 1977.

The first two objectives apply specifically to the First Normal Form; the last two apply to all normal forms. These terms will be defined shortly.

The entire normalization process is based upon the analysis of relations, their schemes, their primary keys and their functional dependencies. Whenever a relation does not meet a normal form test, the relation must be *decomposed* or broken into some other relations that individually meet the criteria of the normal form test. Initially, E. F. Codd proposed three normal forms that he called first, second, and third normal form. These forms are generally abbreviated and referred to as 1NF, 2NF, and 3NF respectively. In addition to these original normal forms there exist others such as the Boyce-Codd Normal Form[3] (BCNF), Fourth Normal Form (4NF), and Fifth Normal Form (5NF). The relationship between some of these normal forms is shown in Fig. 5-1. This figure is sometimes referred to as the normal form "onion". In this book we will only address the following forms: 1NF, 2NF, 3NF, and BCNF. A discussion of these forms follows.

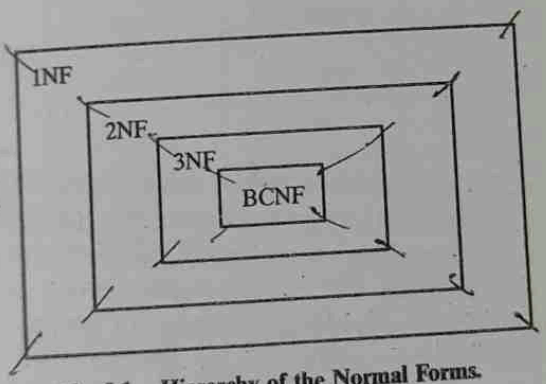


Fig. 5-1. Hierarchy of the Normal Forms.

5.2 First Normal Form

Sometimes, during the process of designing a database it may be necessary to transform into a relation a given table that in some of its entries (the intersection of a row and a column) may have more than one value. For example, consider the PROJECT table shown in Fig. 5-2 where one or more employees may be assigned to a project. Notice that for each Project id (Proj-ID) every "row" of the table has more than one value under the columns Emp-ID, Emp-Name, Emp-Dpt, Emp-Hrly-Rate, and Total-Hrs.

---

[3] ...ly proposed in E. F. Codd; "Recent Investigations into Relational Data Base Systems," *Proc. IFIP* ...Stockholm, Sweden, 1974.

Proj-ID	Proj-Name	Proj-Mgr-ID	Emp-ID	Emp-Name	Emp-Dpt	Emp-Hrly-Rate	Total-Hrs
100	E-commerce	789487453	123423479 980808980 234809000 542298973	Heydary Jones Alexander Johnson	MIS TechSupport TechSupport TechDoc	65 45 35 30	10 6 6 12
110	Distance-Ed	820972445	432329700 689231199 712093093	Mantle Richardson Howard	MIS TechSupport TechDoc	50 35 30	5 12 8
120	Cyber	980212343	834920043 380802233 553208932 123423479	Lopez Harrison Olivier Heydary	Engineering TechSupport TechDoc MIS	80 35 30 65	4 11 12 07
130	Nitts	550227043	340783453	Shaw	MIS	65	07

Fig. 5-2. The PROJECT table.

To refer to this type of table and how tables relate to relations some new terminology is necessary. Table entries that have more than one value are called *multivalue* entries. Tables with multivalue entries are called *unnormalized* tables. Within an unnormalized table, we will call a *repeating group* an attribute or group of attributes that may have multivalue entries for single occurrences of the table identifier. This last term refers to the attribute that allows us to distinguish the different rows of the unnormalized table. Using this terminology we can describe the PROJECT table shown above as an unnormalized table where attributes Emp-ID, Emp-Name, Emp-Dpt, Emp-Hrly-Rate, and Total-Hrs are repeating groups. As we indicated before in Section 2.1, this type of table cannot be considered a relation because there are entries with more than one value.

To be able to represent this table as a relation and to implement it in a RDBMS, it is necessary *to normalize the table*. In other words, we need to put the table in first normal form. We can formally define the latter term as follows: A relation r(R) is said to be in *First Normal Form (1NF)* if and only if every entry of the relation (the intersection of a tuple and a column) has at most a single value. Some authors prefer to say that a relation is in 1NF if and only if all its attributes are based upon a simple domain. These two definitions are equivalent. If all relations of a database are in 1NF we will say that the database is in 1NF.

The objective of normalizing a table is to remove its repeating groups and ensure that all entries of the resulting table have at most a single value. The reader should be aware that by simply removing their repeating groups unnormalized tables do not become relations automatically. Some further manipulation of the resulting table(s) may be necessary to ensure that they are indeed relations. In general, there are two basic approaches to normalize tables. We will consider these two approaches next.

The first approach, known as "flattening the table", removes repeating groups by filling in the "missing" entries of each "incomplete row" of the table with copies of their corresponding nonrepeating attributes. The following example illustrates this.

Example 5.1
Flatten the table of Fig. 5-2. Is the resulting table a relation? If not, how can you transform it to a 1NF relation?

In the PROJECT table, for each individual project, under the Emp-ID, Emp-Name, Emp-Dpt, Emp-Hrly-Rate, and Total-Hrs attributes there is more than one value per entry. To normalize this table, we just fill in the remaining entries by copying the corresponding information from the nonrepeating attributes. For instance, for the row that contains the employee Jones, we fill in the remaining "blank" entries by copying the values of the Proj-ID, Proj-Name, and Proj-Mgr-ID columns. This row has now a single value in each of its entries. In the new table shown below, we have grayed all the new set of tuples for the employees of the E-commerce project. We have repeated a similar process for the employees of the remaining two projects. The normalized representation of the PROJECT table is:

Proj-ID	Proj-Name	Proj-Mgr-ID	Emp-ID	Emp-Name	Emp-Dpt	Emp-Hrly-Rate	Total-Hrs
100	E-commerce	789487453	123423479	Heydary	MIS	65	10
100	E-commerce	789487453	980808980	Jones	TechSupport	45	6
100	E-commerce	789487453	234809000	Alexander	TechSupport	35	6
100	E-commerce	789487453	542298973	Johnson	TechDoc	30	12
110	Distance-Ed	820972445	432329700	Mantle	MIS	50	5
110	Distance-Ed	820972445	689231199	Richardson	TechSupport	35	12
110	Distance-Ed	820972445	712093093	Howard	TechDoc	30	8
120	Cyber	980212343	834920043	Lopez	Engineering	80	4
120	Cyber	980212343	380802233	Harrison	TechSupport	35	11
120	Cyber	980212343	553208932	Olivier	TechDoc	30	12
120	Cyber	789487453	123423479	Heydary	MIS	65	10
130	Nitts	550227043	340783453	Shaw	Cabling	40	27

This normalized PROJECT table *is not* a relation because it does not have a primary key. The attribute Proj-ID no longer identifies uniquely any row. Notice that all rows in the grayed area have the same Proj-ID. To transform this table into a relation a primary key needs to be defined. A suitable PK for this table is the composite key (Proj-ID, Emp-ID). Observe that any other combination of the attributes of the table will not work as a PK.

The second approach for normalizing a table requires that the table be decomposed into two new tables that will replace the original table. Decomposition of a relation involves separating the attributes of the relation to create the schemes of two new relations. However, before decomposing the original table it is necessary to identify an attribute or a set of its attributes that can be used as table identifiers. Assuming that this is the case, one of the two tables contains the table identifier of the original table and *all the nonrepeating* attributes. The other table contains a copy of the table identifier and *all the repeating* attributes. To transform these tables in relations, it may be necessary to identify a PK for each table. If one of the tables has more than one repeating group or if the repeating groups have other repeating groups within themselves this process can be repeated as many times as necessary. The tuples of the new relations are the projection of the original relation into their respective schemes. The following example illustrates this second approach for normalizing tables.

Example 5.2

Normalize the table of Fig. 5-2 using the second approach of normalization.

To normalize the PROJECT table we need to replace it by two new tables. The first table contains the table attribute and the nonrepeating groups. These attributes are: Proj-ID (the table identifier), Proj-Name, and Proj-Mgr-ID. The second table contains the table identifier and all the repeating groups. Therefore, the attributes of this table are: Proj-ID, Emp-ID, Emp-Name, Emp-Dpt, Emp-Hrly-Rate, and Total-Hrs. To transform the latter table into a relation, it is necessary to assign it a PK. These two new 1NF relations are shown below. Notice that for the PROJECT-EMPLOYEE table the composite attribute (Proj-ID, Emp-ID) is an appropriate PK.

PROJECT

Proj-ID	Proj-Name	Proj-Mgr-ID
100	E-commerce	789487453
110	Distance-Ed	820972445
120	Cyber	980212343

PROJECT-EMPLOYEE

Proj-ID	Emp-ID	Emp-Name	Emp-Dpt	Emp-Hrly-Rate	Total-Hrs
100	123423479	Heydary	MIS	65	10
100	980808980	Jones	TechSupport	45	6
100	234809000	Alexander	TechSupport	45	6
100	542298973	Johnson	TechDoc	30	12
110	432329700	Mantle	MIS	65	5
110	689231199	Richardson	TechSupport	45	12
110	712093093	Howard	TechDoc	30	8
120	834920043	Lopez	Engineering	80	4
120	380802233	Harrison	TechSupport	45	11
120	553208932	Olivier	TechDoc	30	12
120	123423479	Heydary	MIS	65	10
130	340783453	Shaw	Cabling	40	27

At this point the reader may ask which of these two approaches is better to use. Actually, both approaches are correct because they transform any unnormalized table into a 1NF relation. However, the authors consider the second approach more efficient because the relations produced are less redundant. In addition, as we will see in the next section, the single table obtained using the first approach will be eventually broken into the same two tables obtained in the second approach.

5.3 Data Anomalies in 1NF Relations

Redundancies in 1NF relations lead to a variety of *data anomalies*. By this latter term we mean side effects that the data experience as a result of some relational operations. Data anomalies are divided into three general categories: *insertion*, *deletion* and *update* anomalies. They are named respectively after the relational operations of INSERT, DELETE and UPDATE because it is during the application of these operations that a relation may experience anomalies. In reality there are only two types of anomalies: update and insertion/deletion anomalies. The latter category can be considered as only one category because

a relation that experiences deletion anomalies will also have insertion anomalies. One cannot exist without the other. However, for explanation purposes we will consider these data anomalies as divided into three categories.

To help us understand the concept of data anomalies in 1NF relations, let's consider the functional dependency EMP-ID → EMP-DPT of the PROJECT-EMPLOYEE relation of the previous section. *Insertion anomalies* occur in this relation because we cannot insert information about any new employee that is going to work for a particular department unless that employee is already assigned to a project. Remember that the composite key of this relation is (Proj-ID, Emp-ID). Notice also that the integrity constraint prevents any attribute of a composite key from being NULL. Likewise, the relation experiences *deletion anomalies* whenever we delete the last tuple of a particular employee. In this case, we not only delete the project information that connects that employee to a particular project but also lose other information about the department for which this employee works. Consider the information that is lost if employee Shaw is deleted. In addition to these two types of anomalies, the relation is also susceptible to *update anomalies* because the department for which an employee works may appear many times in the table. It is this redundancy of information that causes the anomaly because if an employee moves to another department, we are now faced with two problems: we either search the entire table looking for that employee and update his or her Emp-Dpt value or we miss one or more tuples of that employee and end up with an inconsistent database. For small tables, this type of anomaly may not seem to be much of a problem, but it is easy to imagine situations where there may be thousands of tuples that experience similar anomaly.

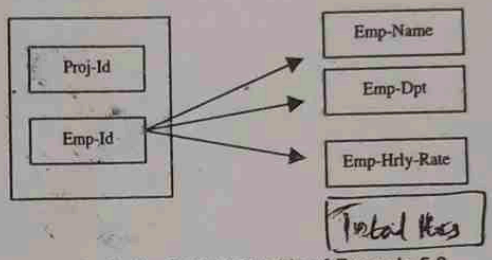
5.4 Partial Dependencies

Given a relation r(R), the sets of attributes X and Y (X, Y ⊂ R), and X → Y, we will say that attribute *Y is fully dependent on attribute X* if and only if there is no proper subset W of X such that W → Y. If there is a proper subset W of X such that W → Y then attribute Y is said to be *partially dependent on attribute X*. Another way of expressing the concept of partial and full dependency is as follows: Given $A_1A_2A_3\ldots A_m \rightarrow B_1B_2\ldots B_n$ with (m > n), the set of attributes $B_1B_2\ldots B_n$ is said to be partially dependent on attributes $A_1A_2A_3\ldots A_m$ if and only if there exists a proper subset of attributes $A_cA_dA_e\ldots A_k \subset A_1A_2A_3\ldots A_m$ such that $A_cA_dA_e\ldots A_k \rightarrow B_1B_2\ldots B_n$. In other words, attributes $B_1B_2\ldots B_n$ are partially dependent on the determinant if there is a subset of the attributes of the determinant that functionally determine the right-hand side of the FD. If no such a subset of attributes of the determinant exist then we say that attributes $B_1B_2\ldots B_n$ are fully dependent on the determinant of the FD.

The identification of partial dependencies is a critical aspect of transforming relations to 2NF as we will see in Section 5.5. Example 5.3 illustrates this concept of partial dependency.

Example 5.3
Identify any partial dependencies in the PROJECT-EMPLOYEE relation.

As indicated before, the PK of this relation is formed by the attributes Proj-ID and Emp-ID. This implies that *Proj-ID, Emp-ID* functionally determines any individual attribute or any combination of attributes of the relation. However, we only need attribute Emp-ID to functionally determine the following attributes: Emp-Name, Emp-Dpt, Emp-Hrly-Rate, and Total-Hrs. In other words, attributes Emp-Name, Emp-Dpt, and Emp-Hrly-Rate are partially dependent on the key. A diagram representing the partial dependency of these attributes on the composite key is shown below.



Example 5.4
Find the partial dependencies in the PROJECT table of Example 5.2.

There are no partial dependencies in this table because the determinant of the key only has a single attribute.

5.5 Second Normal Form

A relation r(R) is in *Second Normal Form* (2NF) if and only if the following two conditions are met simultaneously:

(1) r(R) is already in 1NF.

(2) No nonprime attribute is partially dependent on any key or, equivalently, each nonprime attribute in R is fully dependent upon every key (including candidate keys).

Notice that in order to find the nonprime attributes of R we need to identify *all* prime attributes of R. As a consequence of this we need to identify first *all* possible keys of the relation. The nonprime attributes are then calculated as R – P where P is the set of all prime attributes and R is the relational scheme of R.

If all relations of a database are in 2NF we will say that the database is in 2NF.

To transform a relation into a 2NF relation we will follow the approach illustrated in Fig. 5-3. In this diagram, the prime attributes are indicated with asterisks and functional dependencies with arrows. The composite key is indicated with a curly bracket.

Adapted from *Computer Data-Base Organization*, 2nd edn, by James Martin, Prentice-Hall, 1975.

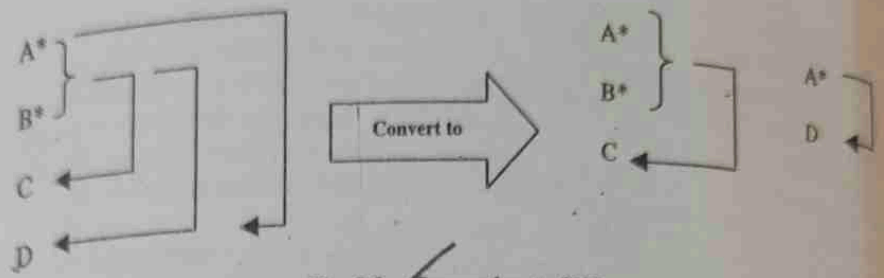


Fig. 5-3. Conversion to 2NF.

Notice that in Fig. 5-3, the PK consists of attributes A and B. These two attributes determine all other attributes. Attributes A and B are the only prime attributes. Attribute C is fully dependent on the key. Attribute D is partially dependent on the key because we only need attribute A to functionally determine it. Attributes C and D are nonprime. Observe that in the diagram the original relation gets replaced by two new relations. The first new relation has three attributes: A, B, and C. The PK of this relation is AB (the PK of the original relation). The second relation has A and D as its only two attributes. Observe that attribute A has been designated as the PK of the second relation and that attribute D is now fully dependent on the key. Although the diagram only shows four attributes, we can generalize this procedure for any relation that we need to transform to 2NF if we assume that C stands for the collection of attributes that are fully dependent on the key and D stands for the collection of attributes that are partially dependent on the key. The next example illustrates how to transform a relation into 2NF using this general procedure.

Example 5.5

Transform the PROJECT-EMPLOYEE relation into a 2NF relation.

The general procedure calls for breaking this relation into two new relations. The first relation has as its PK the PK of PROJECT-EMPLOYEE (Proj-ID, Emp-ID), and the remaining attributes of this relation are all the attributes that fully depend on this key. In this case, the only attribute that fully depends on this composite key is Total-Hours. The scheme of this new relation that we have named HOURS-ASSIGNED is as follows:

HOURS-ASSIGNED (Proj-ID, Emp-ID, Total-Hours)

The second relation contains as its key the Emp-ID attribute since this attribute fully determines the Emp-Name, Emp-Dpt, and Emp-Hrly-Rate. The scheme of this relation is as follows:

EMPLOYEE (Emp-ID, Emp-Name, Emp-Dpt, Emp-Hrly-Rate)

5.6 Data Anomalies in 2NF Relations

Relations in 2NF are still subject to data anomalies. For sake of explanation, let us assume that the department in which an employee works functionally determines the hourly rate charged by that employee. That is, Emp-Dpt → Emp-Hrly-Rate. This fact was not considered in the explanation of the previous normal form but it is not an unrealistic situation. *Insertion anomalies* occur in the EMPLOYEE relation. For example, consider a situation where we would like to set in advance the rate to be charged by the employees of a new department. We cannot insert this information until there is an employee assigned to that department. Notice that the rate that a department charges is independent of whether or not it has employees. The EMPLOYEE relation is also susceptible to *deletion anomalies*. This type of anomaly occurs whenever we delete the tuple of an employee who happens to be the only employee left in a department. In this case, we will also lose the information about the rate that the department charges. *Update anomalies* will also occur in the EMPLOYEE relation because there may be several employees from the same department working on different projects. If the department rate changes, we need to make sure that the corresponding rate is changed for all employees that work for that department. Otherwise, the database may end up in an inconsistent state.

5.7 Transitive Dependencies

Assume that A, B, and C are the set of attributes of a relation r(R). Further assume that the following functional dependencies are satisfied simultaneously: A → B, B ↛ A, B → C, and C ↛ A and A → C. Observe that C → B is neither prohibited nor required. If all these conditions are true, we will say that attribute *C is transitively dependent on attribute A*. It should be clear that these FDs determine the conditions for *having* a transitive dependency of attribute C on A. If any of these FDs are not satisfied then attribute C is not transitively dependent on attribute A. The diagram shown in Fig. 5-4 summarizes these conditions. In this diagram the arrows are equivalent to the symbol "→" that we use for denoting FDs. Notice that the functional dependency A → C may not be explicitly indicated but it holds true due to the Transitivity axiom. The requirements that B ↛ C and C ↛ A are necessary to ensure that attributes A and B are nonprime attributes.

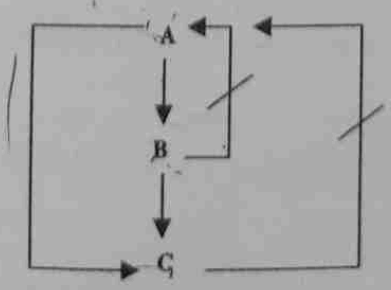


Fig. 5-4. Conditions that define the transitive dependency of attribute C on attribute A.

5.8 Third Normal Form

A relation $t(R)$ is in Third Normal Form (3NF) if and only if the following conditions are satisfied simultaneously:

(1) $t(R)$ is already in 2NF.
(2) No nonprime attribute is transitively dependent on the key.

The reader should not get confused with the conditions stated in Fig. 5-4 and the second condition of the definition of 3NF. Notice that Fig. 5-4 states the conditions that need to be met so that the nonprime attribute C can be transitively dependent on key A. The definition of 3NF requires that these conditions are not met if A is the key attribute and C is a nonprime attribute.

Another way of expressing the conditions for Third Normal Form is as follows:

(1) $t(R)$ is already in 2NF.
(2) No nonprime attribute functionally determines any other nonprime attribute.

Since these two sets of conditions are equivalent (see Solved Problem 5.5), we will use the one that is most convenient for the particular problem at hand.

As these two definitions of 3NF imply, the objective of transforming relations into 3NF is to remove all transitive dependencies. To transform a 2NF relation into a 3NF we will follow the approach indicated by Fig. 5-5. In this figure, assume that any FD not implicitly indicated does not hold. An asterisk indicates the key attribute and the arrows denote functional dependencies. The dashed line indicates that the FD $A \rightarrow C$ may not be explicitly given but it is always present because it can be derived using the inference axioms.

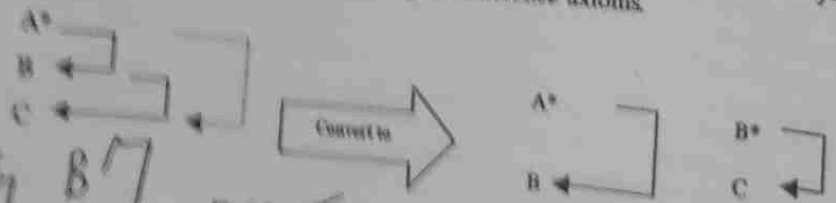


Fig. 5-5 Conversion to Third Normal Form.

The next example illustrates how to transform a relation into 3NF using this general procedure.

Example 5.6

Convert to 3NF the EMPLOYEE relation of Example 5.5 using the first definition of 3NF.

The relation EMPLOYEE of Example 5.5 is not in 3NF because there is a transitive dependency of a nonprime attribute on the primary key of the relation. In this case, the nonprime attribute Emp-Hrly-Rate is transitively dependent on

the key through the functional dependency Emp-Dpt → Emp-Hrly-Rate. Notice that all other conditions required by the definition are met by this set of FDs. In particular, we have that Emp-Dpt ↛ Emp-ID and Emp-Hrly-Rate ↛ Emp-ID. To transform this relation into a 3NF relation, it is necessary to remove any transitive dependency of a nonprime attribute on the key. According to the diagram of Fig. 5-5, it is necessary to create two new relations. The scheme of the first relation is EMPLOYEE (Emp-ID, Emp-Name, Emp-Dpt). The scheme of the second relation is CHARGES (Emp-Dpt, Emp-Hrly-Rate). Observe that in the second relation, the Emp-Dpt attribute has been made the PK of the relation as required by the diagram.

Example 5.7
Using the second definition of 3NF, how can we determine that the EMPLOYEE relation of Example 5.5 is not in 3NF? If we use the second definition for 3NF, do we need to use a different procedure to transform the relation to 3NF?

We can determine that the relation is not in 3NF by noticing that Emp-Dpt → Emp-Hrly-Rate and both attributes are nonprime. To transform this relation to 3NF we use the same general procedure of Fig. 5-5.

The new set of relations that we have obtained through this normalization process does not exhibit any of the anomalies of the previous forms. That is, we can insert, delete and update tuples without any of the side effects that were present in 1NF and 2NF.

5.9 Data Anomalies in 3NF Relations

The Third Normal Form helped us to get rid of the data anomalies caused either by transitive dependencies on the PK or by dependencies of a nonprime attribute on another nonprime attribute. However, relations in 3NF are still susceptible to data anomalies particularly when the relations have two overlapping candidate keys or when a nonprime attribute functionally determines a prime attribute. The following two examples will illustrate this.

Example 5.8
Consider the CERTIFICATION-PROGRAM (Area, Course, Section, Time, Location)

Area	Course	Section	Time	Location
East Coast	SQL 101	Introduction	8:00-10:00	Atlanta Educational Center
East Coast	SQL 101	Intermediate	10:00-12:00	New York Educational Center
West Coast	SQL 101	Advanced	8:00,-10:00	Los Angeles Educational Center

This relation is in 3NF because neither of the nonprime attributes functionally determines the other attributes. However, if there is only one instructional center per city then we will have that Location → Area. This dependency does not violate the 3NF condition but there are some anomalies in the data. For example, assume that if we delete the last tuple of the relation we may lose information about the location of the educational center.

Example 5.9

Consider the MANUFACTURER relation shown below where each manufacturer has a unique ID and name. Manufacturers produce items (identified by their unique item numbers) in the amounts indicated. Manufacturers may produce more than one item and different manufacturers may produce the same items.

MANUFACTURER

ID	Name	Item-No	Quantity
M101	Electronics USA	H3552	1000
M101	Electronics USA	J08732	500
M101	Electronics USA	Y23490	200
M322	Electronics-R-Us	H3552	900

This MANUFACTURER relation has two candidate keys: (ID, Item-No) and (Name, Item-No) that overlap on the attribute Item-No. The relation is in 3NF because there is only one nonprime attribute and therefore it is impossible that this attribute can determine another nonprime attribute.

The relation MANUFACTURER is susceptible to update anomalies. Consider for example the case in which one of the manufacturers changes its name. If the value of this attribute is not changed in all of the corresponding tuples there is the possibility of having an inconsistent database.

5.10 Boyce-Codd Normal Form

To eliminate these anomalies in 3NF relations, it is necessary to carry out the normalization process to the next higher step, the Boyce-Codd Normal Form. This concept is formalized next.

A relation r(R) is in Boyce-Codd Normal Form (BCNF) if and only if the following conditions are met simultaneously:

(1) The relation is in 1NF.

(2) For every functional dependency of the form $X \rightarrow A$, we have that either $A \subset X$ or X is a superkey of r. In other words, every functional dependency is either a trivial dependency or in the case that the functional dependency is not trivial then X must be a superkey.

Notice that the definition of BCNF does not make any reference to the concepts of full or partial dependency. However, from this definition we can make the following assertions about the prime and nonprime attributes of the relational scheme:

- All nonprime attributes must be fully dependent on every key.
- All prime attributes must be fully dependent on all keys of which they are not part.

In Fig. 5-1, we can observe that the set of 3NF relations are a proper subset of the BCNF relations. That is, all BCNF are in 3NF but not all 3NF are in BCNF. In this sense, the definition of BCNF is said to be more restrictive. The following example illustrates this.

Example 5.10
Using the MANUFACTURER relation of the previous example, transform it to BCNF.

To transform this relation into BCNF we can decompose it into the following set of relational schemes:

Set No. 1
MANUFACTURER(ID, Name)
MANUFACTURER-PART(ID, Item-No, Quantity)
or
Set No. 2
MANUFACTURER(ID, Name)
MANUFACTURER-PART(Name, Item-No, Quantity)

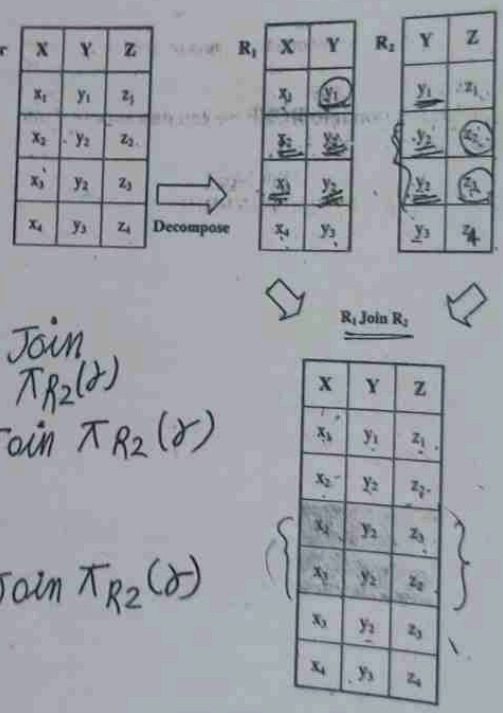
Notice that both relations are in BCNF and that the update data anomaly is no longer present. In this example, we have decomposed the relation in a manner that is convenient for purpose of the explanation. In Solved Problem 5.16 we will consider a more systematic procedure to transform relations into BCNF.

5.11 Lossless or Lossy Decompositions

So far we have decomposed a relation using either a general procedure (Sections 5.5 and 5.8) or an ad-hoc method (Section 5.10). However, whenever a relation is decomposed it is necessary to ensure that the data in the original relation is represented faithfully by the data in the relations that are the result of the decomposition; that is, we need to make sure that we can recover the original relation from the new relations that have replaced it. Generally the original relation is recovered by forming the natural join of the new relations. If we can recover the original relation we say that the decomposition is *lossless* with respect to D or that the relation has a lossless-join decomposition with respect to D where D is a set of FDs satisfied by the original relation. If the relation cannot be recovered we say that the decomposition is *lossy*. We can formalize this concept as follows:

Assume that a relation r(R) has been replaced by a collection or relation $r_1(R_1)$, $r_2(R_2), \ldots r_n(R_n)$ such that $R = R_1 \cup R_2 \cup \ldots \cup R_n$ and D is a set of dependencies satisfied by r. We say that the decomposition is *lossless with respect to D* or is a *lossless-join decomposition with respect to D* if and only if $r = \pi_{R1}(r)$ join $\pi_{R2}(r)$ join ... join $\pi_{Rn}(r)$. That is, the relation r is lossless if it is the natural join of its projections onto the R_i's. If the relation *cannot* be recovered from the natural projection the *decomposition is said to be lossy with respect to D*. The following example illustrates this concept.

Example 5.11
Consider the relation r shown below and its decomposition R_1 and R_2. Assume that $X \rightarrow Y$ and $Z \rightarrow Y$. Is this decomposition lossless or lossy?



$\pi_{R1}(r)$ Join $\pi_{R2}(r)$

$\pi_{R1}(r)$ Join $\pi_{R2}(r)$

$\pi_{R1}(r)$ Join $\pi_{R2}(r)$

Notice that the natural join of relations R_1 and R_2 has tuples that were not present in the original relations. These additional tuples that were not in the original relation are called *spurious tuples* because they represent spurious or false information that is not valid. We have grayed the spurious tuples in the join relation. Since the natural join of relation R_1 and R_2 does not recover the original relation the decomposition is lossy.

5.11.1 TESTING FOR LOSSLESS JOINS

As indicated before, a lossless-join decomposition is necessary to ensure that the relation is recoverable. Determining whether or not a decomposition is lossless or lossy with respect to a set of FDs is a fairly easy procedure if we use the Lossless Join Algorithm. This algorithm is shown next.

The Lossless-Join Algorithm

The algorithm has two inputs. The first input is a set of relations $r_1(R_1)$, $r_2(R_2), \ldots, r_n(R_k)$ that have replaced a relation $r(A_1, A_2, A_3, \ldots, A_n)$ where $R = [A_1, A_2, A_3, \ldots, A_n] = R_1 \cup R_2 \cup \ldots \cup R_k$. The second input to the algorithm is a set F of functional dependencies satisfied by r. The output of the algorithm is a decision stating that the decomposition is lossless or lossy. To apply this algorithm proceed as follows:

(1) Construct a table with n columns (n is the number of attributes of the original relation) and k rows (k is the number of relations in which the original relation has been decomposed). Label the columns of this table $A_1, A_2, A_3, \ldots, A_n$ respectively. Label the rows $R_1, R_2, R_3, \ldots, R_k$.

(2) Fill in the entries of this table as follows:
For each attribute A_i check if this attribute is one of the attributes of the scheme of the relation R_j. If attribute A_i is in the scheme of R_j, then in the entry (A_i, R_j) of the table write a_i. If attribute A_i *is not* one of the attributes in the scheme of relation R_j then in the entry (A_i, R_j) write b_{ij}.

(3) For each of the functional dependencies $X \rightarrow Y$ of F do the following *until* it is not possible to make any more changes to the table. When the table no longer changes continue with step 4.

Look for two or more rows that have the same value under the attribute or attributes that comprise X (the determinant of the FD under consideration). There are two possible outcomes to this search:

(3-a) If there are two or more rows with the same value under the attribute or attributes of the determinant X then make equal their entries under attribute Y (the right-hand side of the FD under consideration). When making equal two or more symbols under any column, if one of them is a_j, make all of them a_j. If they are b_{ij} and b_{kj}, choose one of these two values as the representative value and make the other values equal to it. Continue with step 3.

(3-b) If there are no two rows with the same value under the attribute or attributes of the determinant X continue with step 3.

(4) Check the rows of the table. If there is a row with its entries equal to $a_1 a_2 \ldots a_n$ then the decomposition is lossless. Otherwise, the decomposition is lossy.

Example 5.12[1]
Consider the relation r(X, Y, Z, W, Q), the set F = {X → Z, Y → Z, Z → W, WQ → Z, ZQ → X} and the decomposition of r into relations R_1(X, W), R_2(X, Y), R_3(Y, Q), R_4(Z, W, Q), and R_5(X, Q). Using the Lossless-Join Algorithm determine if the decomposition is lossless or lossy.

Steps (1–2) Since the original relation has 5 attributes and the original relation has been decomposed into 5 relations, we need to create a table with 5 columns and 5 rows. The columns of the table are named X, Y, Z, W and Q respectively. The rows of the table are named R_1, R_2, R_3, R_4 and R_5 respectively. For explanation purposes we have renamed the attributes A, $A_2, \ldots, A_5$. The table is shown below.

	X(A_1)	Y(A_2)	Z(A_3)	W(A_4)	Q(A_5)
R_1	a_1				
R_2					
R_3					
R_4					
R_5					

Since X is one of the attributes of relation R_1 we write a_1 in the entry (R_1, A_1) or its equivalent (R_1, X). In other words, under column A_1 (or X) and row R_1 we write a_1 as seen in the row pointed to by the arrow (see below). Likewise, we write a_4 in the entry corresponding to (R_1, A_4) or its equivalent (R_1, W) because W is also an attribute of relation R_1. That is, under column A_4 (or W) and row R_1 we write a_4 as seen in the row pointed to by the arrow (see below). In the remaining entries of row R_1 we write the values b_{ij} where i is the column number and j is the row number. These entries (from left to right) are b_{12}, b_{13} and b_{15} respectively as seen in the row pointed to by the arrow in the following table:

	X(A_1)	Y(A_2)	Z(A_3)	W(A_4)	Q(A_5)
→ R_1	a_1	b_{12}	b_{13}	a_4	b_{15}
R_2					
R_3					
R_4					
R_5					

[1] A similar example appears in *Principles of Database and Knowledge-Base Systems* Vol 1, by J. D. Ullman, Computer Scien

Proceeding in this fashion we fill in the remaining entries in the table. After filling in all entries the table looks like this.

	$X(A_1)$	$Y(A_2)$	$Z(A_3)$	$W(A_4)$	$Q(A_5)$
R_1	a_1	b_{12}	b_{13}	a_4	b_{15}
R_2	a_1	a_2	b_{23}	b_{24}	b_{25}
R_3	b_{31}	a_2	b_{33}	b_{34}	a_5
R_4	b_{41}	b_{42}	a_3	a_4	a_5
R_5	a_1	b_{52}	b_{53}	b_{54}	a_5

Steps (3/3-a) 1st time

Considering $X \rightarrow Z$ or its equivalent $A_1 \rightarrow A_3$ we look for tuples that have the same value under column X (or A_1). In this case, rows R_1, R_2 and R_5 have the same value a_1. Therefore, we equate or make equal the values under attribute Z (or A_3). Remember that Z (or A_3) is the right-hand side of the functional dependency that we are considering. Notice that the corresponding entries for rows R_1, R_2 and R_5 under the Z column are b_{13}, b_{23} and b_{53} respectively. To equate these entries to the same value we need to choose one of them arbitrarily and make the other two entries the same. Choosing b_{13} as the representative value and changing the remaining two entries to this value, we have that the table looks like the one shown next. Notice that we have grayed the entries that were made equal to the value b_{13} of row R_1. Repeat step 3.

	$X(A_1)$	$Y(A_2)$	$Z(A_3)$	$W(A_4)$	$Q(A_5)$
R_1	a_1	b_{12}	b_{13}	a_4	b_{15}
R_2	a_1	a_2	b_{13}	b_{24}	b_{25}
R_3	b_{31}	a_2	b_{33}	b_{34}	a_5
R_4	b_{41}	b_{42}	a_3	a_4	a_5
R_5	a_1	b_{52}	b_{13}	b_{52}	a_5

Steps (3/3-a) 2nd time

Considering $Y \rightarrow Z$ or its equivalent $A_2 \rightarrow A_3$, we now look for rows that have the same value in the column Y (or A_2). In this case, rows R_2 and R_3 have the same value a_2. Therefore, we need to equate the corresponding entries under column Z (or A_3). Choosing b_{13} as the representative value we can change

the b_{13} entry to b_{13}. After this change the table looks like the one shown below. Notice that we have grayed the row whose entry changed. Repeat step 3.

	$X(A_1)$	$Y(A_2)$	$Z(A_3)$	$W(A_4)$	$Q(A_5)$
R_1	a_1	b_{12}	b_{13}	a_4	b_{15}
R_2	a_1	a_2	b_{13}	b_{24}	b_{25}
R_3	b_{31}	a_2	b_{13}	b_{34}	a_5
R_4	b_{41}	b_{42}	a_3	a_4	a_5
R_5	a_1	b_{52}	b_{13}	b_{54}	a_5

Steps (3/3-a) 3rd time

Considering $Z \rightarrow W$ or its equivalent $A_3 \rightarrow A_4$, we look for tuples that have identical values under column Z (or A_3). In this case rows R_1, R_2, R_3 and R_5 have the same value, b_{13}, under this column. Therefore, we need to make equal all entries under column W (or A_4). Since one of the values under column W (or A_4) is a_4 (for row R_1), we make all the b_{i4}'s entries equal to a_4. The resulting table is shown next. As before we have grayed all entries that were changed. Repeat step 3.

	$X(A_1)$	$Y(A_2)$	$Z(A_3)$	$W(A_4)$	$Q(A_5)$
R_1	a_1	b_{12}	a_3	a_4	b_{15}
R_2	a_1	a_2	a_3	a_4	b_{25}
R_3	b_{31}	a_2	a_3	a_4	a_5
R_4	b_{41}	b_{42}	a_3	a_4	a_5
R_5	a_1	b_{52}	a_3	a_4	a_5

Steps (3/3-a) 4th time

Considering $WQ \rightarrow Z$ or its equivalent $A_4A_5 \rightarrow A_3$, we now look for rows that have identical values under columns A_4 and A_5 simultaneously. Rows R_3, R_4 and R_5 have values of a_4 and a_5 under columns WQ (or A_4 and A_5) respectively. Therefore, we need to equate all the corresponding entries for these rows under column Z (or A_3). Since all the values under column Z (or A_3) are already equal no changes are necessary. Repeat step 3.

Steps (3/3-a) 5th time

Considering $ZQ \rightarrow X$ or its equivalent $A_3A_5 \rightarrow A_1$, we look for rows that have equal values under columns ZQ (or A_3A_5) simultaneously. Rows R_3, R_4 and R_5 have values equal to a_3 and a_5 under columns A_3 and A_5 respectively. Therefore, it is necessary to equate their corresponding entries under column X (or A_1). Since one of the entries is a_1 (for row R_5), we make all the remaining b_{i1}'s equal to a_1. The resulting table, after all the corresponding changes are made, looks like the one shown below. As in all cases before, we have grayed the entries that were affected by the changes. Repeat step 3.

	X(A_1)	Y(A_2)	Z(A_3)	W(A_4)	Q(A_5)
R_1	a_1	b_{12}	a_3	a_4	b_{15}
R_2	a_1	a_2	a_3	a_4	b_{25}
R_3	a_1	a_2	a_3	a_4	a_5
R_4	a_1	b_{42}	a_3	a_4	a_5
R_5	a_1	b_{52}	a_3	a_4	a_5

Steps (3/3-a) 6th time

There are no more FDs to consider and the table cannot undergo any more changes. Therefore, we are through with step 3 and move to step 4.

Step (4) 1st time

We now look for a row that has all a_i's. Since row R_3 has become $a_1\,a_2\,a_3\,a_4\,a_5$ (see below) the decomposition is lossless.

	X(A_1)	Y(A_2)	Z(A_3)	W(A_4)	Q(A_5)
R_1	a_1	b_{12}	a_3	a_4	b_{15}
R_2	a_1	a_2	a_3	a_4	b_{25}
R_3	a_1	a_2	a_3	a_4	a_5
R_4	a_1	b_{42}	a_3	a_4	a_5
R_5	a_1	b_{52}	a_3	a_4	a_5

$A_3A_5 \rightarrow A_1$, we look for
olumns ZQ (or A_3A_5)
ave values equal to a_3
ctively. Therefore, it is
entries under column
(for row R_3), we make
esulting table, after all
oks like the one shown
grayed the entries that
tep 3.

			Q(A_6)
			b_{15}
			b_{25}
			a_6
			a_5
			a_5

and the table cannot
e, we are through with

a_i's. Since row R_3 has
composition is lossless.

			Q(A_6)
			b_{15}
			b_{25}
			a_5
			a_5
			a_5

Example 5.13
Consider the relation r(X, Y, Z) and its decomposition $R_1(X, Y)$ and $R_2(Y, Z)$. Assume that $X \rightarrow Y$ and $Z \rightarrow Y$. Use the Lossless-Join Algorithm to determine if this decomposition is lossless or lossy.

Steps (1–2) Since the original relation has 3 attributes and it has been decomposed into 2 relations we need to form a table with three columns and two rows. Notice again that for explanation purposes we have renamed the attributes $A_1, A_2, \ldots, A_5$. However, this step is not necessary.

	$X(A_1)$	$Y(A_2)$	$Z(A_3)$
R_1			
R_2			

The scheme of relation R_1 consists of attributes X and Y. Therefore under columns X (or A_1) and Y (or A_2) we write a_1 and a_2 respectively.

The scheme of relation R_2 consists of attributes Y and Z. Therefore under columns Y (or A_2) and Z (or A_3) we write a_2 and a_3 respectively.

	$X(A_1)$	$Y(A_2)$	$Z(A_3)$
R_1	a_1	a_2	b_{13}
R_2	b_{21}	a_2	a_3

Steps (3/3-b) 1st time Considering $X \rightarrow Y$ we look for rows that have the same value under attribute X. Since there are no two rows with identical value under attribute X, the table remains unchanged and we continue with step 3 again.

Steps (3/3-b) 2nd time Considering $Z \rightarrow Y$ we now look for rows with identical values under attribute Z. Since there are no two rows with identical value under this attribute, the table remains unchanged.

Step (3) 3rd time There are no more FDs to consider in F and therefore the table cannot undergo any more changes. We continue with step 4.

Step (4) 1st time Since there is no row in the table that has all a_i's in its entries the decomposition is lossy. That is, the original table cannot be recovered from the natural join of relations R_1 and R_2.

CHAPTER

As indica
projectio
position
What thi
satisfied
function
straints
preserv
burden
decom
constra
reduce

do

5.12 Preserving Functional Dependencies

As indicated in the previous section, for a relation r to be recoverable from its projections its decomposition must be lossless. In addition to this, the decomposition should satisfy another property known as the *dependency preservation*. What this property requires is that the decomposition satisfy all the FDs that are satisfied by the original relation. The reason this is desirable is that the set of functional dependencies that are satisfied by a relation define integrity constraints that the relation needs to meet. Any decomposition that does not preserve the dependencies of the original relation imposes an unnecessary burden on the RDBMS. If fact, any update to any of the relations of the decomposition would require a join of all these relations to check that the constraints are not violated. This is obviously a time-consuming operation that reduces efficiency of the system. The following example illustrates this.

Example 5.14
Consider a relation r(X, Y, Z) that satisfies the dependencies XY → Z and Z → X. The decomposition of relation r(X, Y, Z) into R_1(XY) and R_2(XZ) is lossless but it does not preserve the dependencies. Consider the following instances of these relations.

R_1

Y	Z
y_1	z_1
y_1	z_2

R_2

Z	X
z_1	x_1
z_2	x_1

Notice that relation R_2 satisfies Z → X but that the join of these two relations does not satisfy the functional dependency XY → Z.

R_1 Join R_2

X	Y	Z
x_1	y_1	z_1
x_1	y_1	z_2

5.12.1 PROJECTION OF A SET OF DEPENDENCIES ONTO A SET OF ATTRIBUTES

To formalize the concept of dependency preservation and to define an algorithm that allows us to test for dependency preservation some additional terminology is necessary. Assume that a relation r(R) has been decomposed into a series of relations $\rho = (R_1, R_2, \ldots, R_k)$ and that F is a set of FDs satisfied by r. We define the *projection of F onto a set of attributes* Z, denoted by $\pi_{(Z)}F$, as follows:

$\pi_{(Z)}F = \{X \rightarrow Y \in F^+ / XY \subset Z\}$ where the symbol "/" reads "such that".

CHAPTER 6

Basic Security Issues

6.1 The Need for Security

In any corporation data is the most valuable resource. Therefore, the database needs to be controlled, managed, protected and secured. In the context of RDBMS we will understand the term *security* to mean the protection of the database against unauthorized access or the intentional or unintentional disclosure, alteration or destruction of the database. In this chapter we consider some of the elementary aspects of data security. We recognize that security concerns are not limited to relational databases. Many of the issues discussed here apply to DBMS with other architectures, as all data must be kept accurate and secure. However, since this book concerns relational databases, in most cases we present a general introduction to the topic of security in relational databases. Also, some specific examples of handling security issues using SQL will be given.

The need for database security began in the early days of computing. In those days, the physical security of the entire computer system was the main concern for the large organizations that could afford the sizeable systems of that time. However, with the proliferation of personal computers, more organizations began relying on databases and logical security of the information itself became more critical. The need for security today is a result of a variety of factors, many mentioned in previous chapters, including:

- Multiple users trying to access and/or change databases at the same or different times.
- More data being kept in single location.
- Databases becoming accessible across communication lines and the existence of distributed databases.
- The advancement of the Internet.
- More specialized software available both to enter the system illegally to extract data and to analyze the information once it is obtained.

Because it is important to centralize database security, the person primarily responsible for the security of the database is usually the Database Administrator (DBA). The DBA must consider a variety of potential threats to the system. Problems may arise from the general public as well as from people within the organization; therefore users must be authenticated and authorized. By *authentication*, we mean that the user has proven to the system that the user really is who he or she claims to be. Often this authentication process takes the form of verifying a password. *Authorization* then involves the specific privileges that the authenticated user has been granted. Authentication and authorization will be discussed in a later section.

Security breaks may be intentional or accidental, so inadvertent changes must be prevented as much as possible by the DBMS itself. Intentional security problems may be malicious or non-malicious, aimed at the computer system itself or at the data contained therein. Most of these issues will be explored in this chapter. The reader should consult other sources for more information. Many books specializing in database security are available, and DBMS documentation will provide specifics on your particular system.

6.2 Physical and Logical Security

Physical security usually means the security of the hardware associated with the system and the protection of the site where the computer resides. *Logical security* encompasses the security measures residing in the operating system and/or the DBMS designed to handle threats to the data, both to its accuracy and also to its confidentiality. One result of the growth of the Internet and the explosion in the number of communication lines is that people do not need to have physical access to the hardware to threaten the data. There is no longer a clear distinction between physical and logical security. This section will discuss some issues specifically relating to physical security of the database. Logical security is far more difficult to accomplish and will be examined throughout the rest of the chapter.

6.2.1 PHYSICAL SECURITY ISSUES

Usually physical security is not the job of the DBA. This job is the responsibility of the security officer in the organization. However, it is prudent for the DBA to make certain that reasonable measures are followed by the organization. In a single user system with the database residing on one personal computer, physical safeguards may entail only locking the workstation and securing the door of the room to limit access. A larger organization might use guards and alarm systems for the rooms that contain the server and other centralized computer equipment. Depending upon the particular organization and the sensitivity of the data, guards and other security measures might also be put into place for the entire building. Most office buildings in urban areas already employ such practices.

Physical threats are not always in the form of illegal entry. Natural events such as fire, floods, and earthquakes must also be considered. Computer equipment should never be kept in a basement area that might fill with water. Special fire extinguishers and smoke alarms are available and should certainly be utilized. Many organizations regularly store backup copies of databases in different cities to allow for fast recovery in the face of massive disasters. The DBA should consult with others in the organization to assess the particular risks and ensure that appropriate procedures are followed.

Example 6.1
A group of three doctors is investigating the physical security of their patient database. Their office is located on the sixth floor in a medical arts complex in a large city. The database is located on a single server and accessible by terminals in each room. What would you suggest they should do?

The doctors need to check to be sure that the building management follows appropriate security measures. If night guards patrol the building, probably secure locks and simple burglar alarms will suffice. If no night guards are available, the doctors could contract with an outside security company for protection. The room containing the server should be locked at all times to prevent patients and other unauthorized personnel from inadvertently entering during the day. The doctors also should consider keeping backup copies of the database at an alternate location that is not in the same building.

6.3 Design Issues

The DBA handles security issues and also all database design. Therefore, the smart DBA will follow a few guidelines to help build the most secure system from the beginning. These guidelines include:

(1) Keep it simple. The smaller and simpler the database design, the fewer ways there are for malicious users to sabotage and for authorized users accidentally to corrupt the data.

(2) Use an open design. All persons accessing the database should understand the schema of the database. (See Chapter 1 to refresh your memory on schemas.) If users thoroughly understand the design, they will be more able to access what they want when they want it with less possibility of error.

(3) Normalize the database. Chapter 5 described the anomalies that occur when the database is not normalized. Normalization takes place during the design phase. It is harder to impose normalization on the relations after the database has been in use.

(4) Always follow the principle of assuming privileges must be explicitly granted rather than excluded. If no privileges are assumed by any user, there is less likelihood that a user will be able to gain illegal access. The designer of the database should help decide which privileges are necessary for each group of users and only grant these privileges. It is safer to err on the side of caution and give the least privilege needed even if a higher level must be granted later rather than to give blanket rights that must be revoked. Section 6.6.6 provides examples of how to grant privileges using SQL.

(5) Create unique views for each user or group of users. Section 6.6.9 describes how to create and maintain views with SQL.

Example 6.2
The DBA of an insurance company is beginning to design a new customer database. What are three possible groups of user? What views should be created for these groups? Which groups would need to be granted rights of access or revision?

Three possible groups of users would be the billing department, the agents, and the customer service personnel. The billing department would need views containing all billing information. They should be able to access and change items such as current balance due and amount paid. The agents need to have a view with complete information on levels and types of coverage. They should be able to change the coverage as requested. Customer service personnel should be able to access both billing information and coverage so that they can answer questions over the telephone. However, they should not be given rights to revise the data.

6.4 Maintenance Issues

Once the database is designed, the DBA's job is not finished. Maintenance is necessary for the lifetime of the database. Most of the security issues regarding maintenance fall into three categories:

(1) Operating System Issues and Availability
(2) Confidentiality and Accountability
(3) Integrity

The following sections examine each of these areas. Once again, it is important to point out that these issues are similar for databases of any type of architecture. Whenever possible, problems and solutions specific to relational databases are discussed. Examples are provided using SQL.

CHAPTER 7

The Entity-Relationship Model

7.1 The Entity-Relationship Model

The modeling, design, and creation of a database is an iterative top-down process. There are several steps in the creation of a database. First, the designer must gather information about the organization in order to ascertain the specific requirements. This usually entails the use of interviews and a thorough examination of all the inputs and outputs of the system. Often software engineering tools, such as data flow diagrams and system flow charts are used. The objective of this step is to determine both the work-flow and the information that is relevant to the organization. Building the Entity-Relationship (E-R) Model is the second step in the process. *The Entity-Relationship (E-R) Model* is a graphical representation of the database logic and includes a detailed description of all entities, relationships, and constraints. After the E-R Diagram has been successfully constructed, the designer creates the Table Instance Charts, one per entity, that contain information about the data types for the attributes of the entities and some sample data. Finally the individual tables are defined and created using a DBMS. The design steps are iterative because changes during a later step may entail revision of the E-R Diagram. Both E-R Diagrams and Table Instance Charts are explained in this chapter. It is important to remember that, as in any top-down approach, these tools should be constructed before beginning to implement the database.

As we explained in Chapter 1, a data model is a way of explaining the logical layout of the data and the relationships of various parts of the database to each other and to the whole. This entire book has focused on the relational database model, where all data are kept in tables or relations that are often connected in some way. By examining the relations alone, we have been able to see the logic

layout. However, we have not yet demonstrated a clear method of illustrating the relationships of all the tables to each other. Just as the blueprints are the key to understanding and creating the design of a building, the E-R Diagram is the key to understanding and creating the design of the database.

The E-R Diagram is an attempt to conceptualize the database. The model helps to ensure that all the client's requirements are met and that these requirements do not conflict with one another. Functional dependencies are also identified. It is at this point that the normalization process explained in Chapter 5 takes place. While the database is still in a conceptual state and independent of any DBMS, it is easy to modify and refine all the individual pieces. Once the model is complete it can then be mapped to any specific type of database software. The model also helps to identify the potential levels of use and to define the views of the data that should be produced.

The E-R Model was described by P. P. Chen in 1976 in a paper "The Entity-Relationship Model: Toward a Unified View of Data."[1] It is now the most widely used model for the design of databases. Since there is not an accepted group of standards, different vendors and authors have developed their own conventions. In this book we will follow the E-R conventions used by the Oracle Corporation.

7.2 Entities and Attributes

The building blocks of the E-R Model are entities, attributes, and relationships. We have already defined entities and attributes in Chapter 1 and have consistently used these concepts throughout the book. Entities are the objects of significance for the organization about which information needs to be known. The characteristics that describe or qualify an entity are the attributes. In the E-R Diagram entities are represented by soft (rounded) boxes with unique names in capital letters. Attributes are the items of information about each entity. Attribute names are unique within the entity and are written in lower-case letters within the box.

Example 7.1
Identify two entities that might be important for a retail business. List at least three attributes for each entity. Then show what the entities and attributes would look like in an E-R Diagram.

Two entities of importance for a business might include employees and customers. Some of the information that may be relevant for employees may be ID number, last name, first name, and salary. Customers would have name, address, city, state, zip, and phone number. Fig. 7-1 shows the sample representation of these entities. Not all attributes are included for space purposes.

[1] Chen, P.O., "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Transactions on Database Systems*, Vol.1, No.1. March 1976.

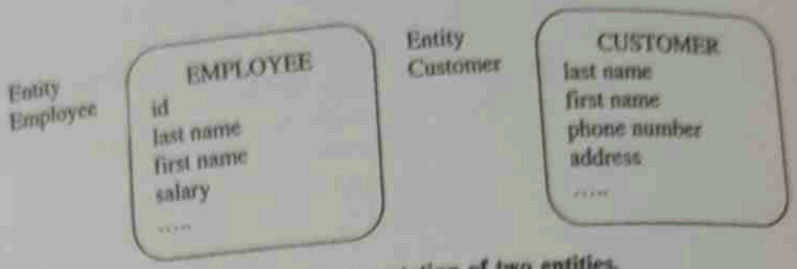


Fig. 7-1. Representation of two entities.

Each entity must have multiple occurrences. For example, the entity EMPLOYEE has one entry for each employee. Each instance has specific values for each attribute. Since entities have multiple occurrences, there must be a way to distinguish one instance from all the others that make up the entity. Therefore each entity must have a Unique Identifier (UID) that is an attribute or set of attributes that uniquely identifies each instance. Chapter 2 explained candidate keys and primary keys. The UID of the entity, which may be a single or composite attribute, is the primary key. If there seems to be no way to identify the entity in a unique way, it should be reevaluated as to whether or not it actually is an entity. The attributes which are the UID are marked in the E-R Diagram with #*. The asterisk (*) indicates the attribute is mandatory, that is it may not be left blank for that instance. The pound sign or hash (#) indicates that the attribute is the UID, or part of the UID. Some attributes are mandatory even though they are not part of the UID, and they would be marked only with an asterisk. Optional attributes may be indicated with an o.

Example 7.2
Choose a UID for the entities EMPLOYEE and CUSTOMER in Example 7.1. Show the new representations.

The obvious UID for EMPLOYEE would be the *ID number*. The entity CUSTOMER could use *name* as UID. However, it is possible for two customers to have the same name and then the instance would no longer be unique. In some cases, it might be possible to use a combination of attributes, such as name and phone number. It is unlikely that two customers with the same name would have the same phone number. However, it is often easier and more beneficial to create a new attribute to be the UID. A new attribute can be added to the entity called ID number to assure that each instance of CUSTOMER will be unique. Notice that the UIDs are marked in Fig 7-2.

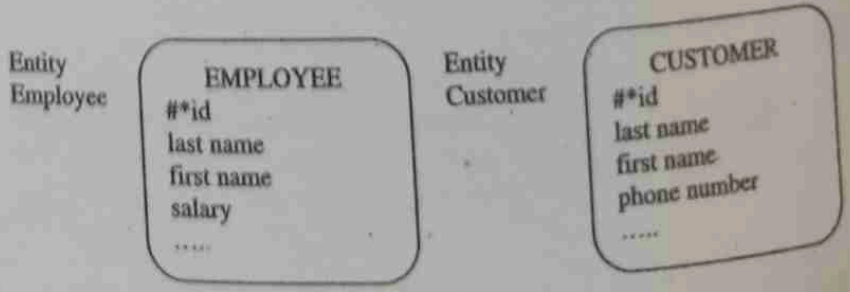


Fig. 7-2. Representation of an entity showing UID.

Remember that there is a difference between the entity and the instance of the entity. The entity is the general category of items, such as products or orders. The instance would be one particular example of the category, such as product #CN1234, a candlestick, which costs $5.14.

7.2.1 IDENTIFYING ENTITIES

Identifying the correct entities for the database is critical. Usually the first step of ascertaining requirements results in some kind of narrative explaining the system. In a narrative, entities are generally represented by nouns. The narrative should be examined closely for significant nouns. Identify synonyms within the list to avoid duplication. Look for meta-words, or words that might be categories rather than instances. Then list as many instances of the entity as you can think of to test that all have the same characteristics and are subject to the same rules. For example, a narrative about a furniture store may include items such as hide-a-bed, sofa, loveseat, or settee. The entity could be "couch" and each of these designations would be a type of couch. All the instances have attributes such as price, color, and type of covering. If entities are found, decide upon names. Establish what information must be kept about each entity. The attributes would be adjectives describing the entity. Decide which attribute or attributes can serve as the UID. Represent the entities that are identified, including the attributes, and mark the UID in the diagram.

Example 7.3
The gathering of specifications for a church database resulted in the explanation shown below. Choose the entities you might want to represent, the attributes for each entity, and model them in a diagram including the UID.

"The Third Presbyterian Church needs to keep track of its members. The members need to receive the newsletter regularly. There are several committees within the church, and each person may serve on only one committee. The Finance Committee wants to record the amount of money that individual members give to the church and report the total to them at the end of the year. The church needs to purchase supplies such as paper and bulletin covers. The secretary, a member of the staff, deals with several different office supply companies."

The nouns that might be identified in this narrative would include MEMBER, VENDOR, and COMMITTEE. Each entity needs a number of attributes that were described in the narrative. Diagrams are shown in Fig. 7-3.

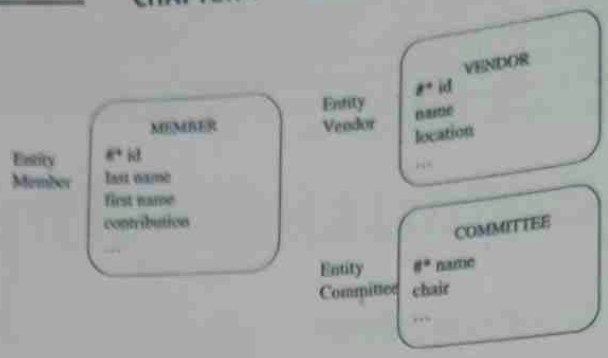


Fig. 7-3. Entities for Third Presbyterian Church.

Not all the necessary attributes are listed in Fig. 7-3 for space purposes. However, each entity has been assigned a UID. The attribute *name* can be used for the entity COMMITTEE because it is safe to assume that the church would not establish two committees with the same name. Both the other entities use ID number as UID. Notice that this diagram has no way to show the committee or committees on which each member serves. The entity MEMBER is related in some way to the entity COMMITTEE. The next section will explore representing such relationships in an E-R Diagram.

7.3 RELATIONSHIPS

Once the entities have been established along with their attributes, it is time to examine relationships. A *relationship* is a two-directional association or connection between two entities. If entities are the nouns of the database, relationships are the transitive verbs. For example, the customer *orders* a product, the member *serves* on a committee.

Begin by designing sentences connecting the entities that might be related. It is important to write out the sentences as clearly as possible. If you end up with a compound sentence, containing a comma or more than one verb, chances are there are two or more entities or relationships involved. If the sentence contains words relating to time, such as first, next, or after, they may be examples of constraints that must be represented in the model. Other types of sentences also reflect relationships. One form is "there are ... X in Y" which can be reworded using a transitive verb. The two sentences represent the two directions of the relationship. For example, "there are passengers on the flight" could be rewritten to reflect the relationship "a flight has passengers." If both flights and passengers are entities then a relationship between those entities has been identified. If the sentence contains an adverb, this often corresponds to an attribute of a relationship.

Once you have identified the entities and attributes, and have written sentences about them to identify relationships, it is time to examine all the

relationships of a particular entity to be sure they are relevant and unique. Remember that a relationship is between two *entities*, and the relationship often goes in both directions. It is important to name the relationships so they can be read in more than one direction. For example, "The product is part of the inventory." and "The inventory is composed of products." Fig. 7-4 shows a list of relationship names that might be helpful in this step. This list was created by Richard Barker.[2]

Useful Pairs of Relationship Names

about	subject of	owned by	owner of*
applicable to	context for*	part of	composed of
at	location of	part of	detailed by
based on	basis for	party to	for
based on	under	party to	holder of
bought in from	supplier of	placed on	responsible for
bound by	for	precluded by	precluded by
change authority for	on	represented by	representation of
classification for	of	responsible for	responsibility of
covered by	for	responsible for	of
defined by	part definition of	run by	carrier for
description of	for *	source of	based on

Note: where the above are marked with an asterisk*, one should only use these as a last resort. For example, "owned by" should only be used as a relationship name when the relationship means legal ownership.

Some of the above names imply the role of a person or organization.

Fig. 7-4. Useful pairs of relationship names.

Each direction of the relationship must have the following:

- a *name*, usually in lower-case letters
- an *optionality*, which is either "mandatory" (continuous line) and is read "must be" or "optional" (dashed line) and is read "may be"
- a *degree* or *cardinality* which is "one or more" indicated by a crowsfoot or "one and only one" indicated by the absence of the crowsfoot

The name is positioned in the diagram close to the entity or noun of the sentence. The optionality is the line connecting the two entity boxes and the degree is indicated at the point where the line meets each box. Fig. 7-5 shows a sample relationship. That relationship can be read left to right as "Each MEMBER *may be* serving on *one or more* COMMITTEES." or right to left as "Each COMMITTEE *may be* made up of *one or more* MEMBERS."

[2] Barker, Richard, *CASE*METHOD*SM *Entity Relationship Modelling*, Addison-Wesley, 1989.

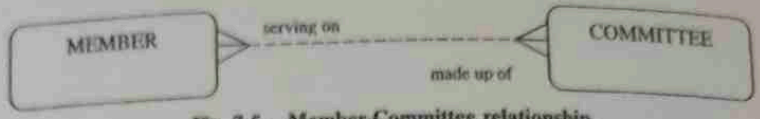
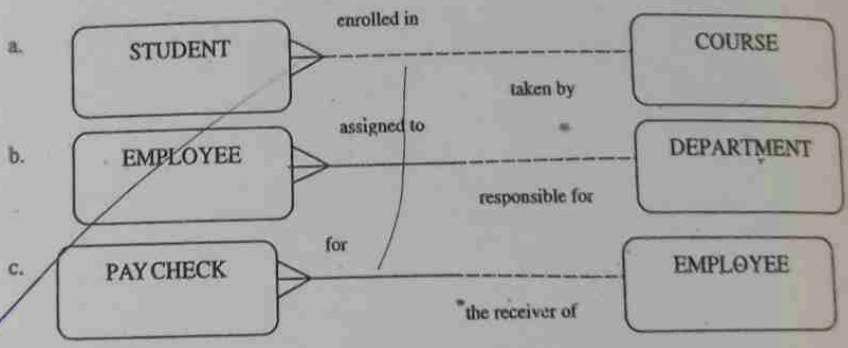


Fig. 7-5. Member-Committee relationship.

It is helpful to follow a few conventions to make a complex diagram easier to read. If one entity has degree of *one or more*, that entity should be on the left or on the top of the chart. Some authors say that "crows fly east or crows fly south." Following this convention, entities that are more volatile, whose instances change frequently, will end up being placed near the left top of the diagram. Entities that are less volatile will end up being placed nearer the bottom.

Example 7.4
Indicate how you would read each of the following relationships. Notice in the chart that the crowsfeet are on the left in each case.



a. L to R: Each STUDENT *may be* enrolled in *one and only one* COURSE.
R to L: Each COURSE *may be* taken by *one or more* STUDENTS.

b. L to R: Each EMPLOYEE *must be* assigned to *one and only one* DEPARTMENT.
R to L: Each DEPARTMENT *may be* responsible for *one or more* EMPLOYEES.

c. L to R: Each PAYCHECK *must be* for *one and only one* EMPLOYEE.
R to L: Each EMPLOYEE *may be* the receiver of *one or more* PAYCHECKS.

...ossible for an entity to have a relationship with itself. In some cases, it is p... ...nship is possible. It would be ... box and indicated in a diagram by a circular line with both ends connected to the Although rare, this kind of recursive relatio... words defining the relationship at each end. For example, if each employee has a supervisor, the supervisor is also an employee.

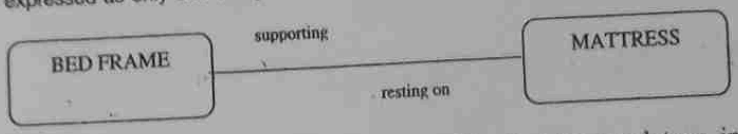
Once you have named the relationship and determined its cardinality and optionality, re-examine the model by reading aloud the relationships between the pairs. Do they make sense, especially for this particular organization? Often the relationships can be verified at this point before more time is invested.

There are three types of relationships: one-to-one, many-to-one, and many-to-many. They will be considered in the next two sections.

7.4 One-to-One Relationships

The *one-to-one relationship* has the cardinality or degree of *one and only one* in both directions. These relationships are denoted by *1 to 1* or *1:1*. 1:1 relationships are rare, especially 1:1 relationships that are mandatory in both directions. If you find a 1:1 relationship, examine the two entities closely, as they may actually be the same entity.

Example 7.5
A relationship exists that can be written two ways: "The bed frame *must* be supporting *one and only one* mattress." "The mattress *must* be resting on *one and only one* bed frame." Diagram this 1:1 relationship. Can the entities be expressed as only one entity?

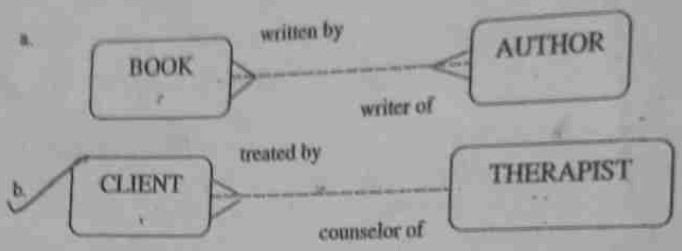


The diagram is shown as a 1:1 relationship that is mandatory in both directions. Since each bed frame supports only one mattress, you should consider the possibility that the two entities are really only one entity BED. The needs of the particular database will suggest whether they should be combined into one.

7.5 Many-to-One and Many-to-Many Relationships

The *many-to-one relationship* has a cardinality in one direction of *one or more* and in the other direction of *one and only one*. This relationship is usually denoted as M:1 or M to 1. It should be obvious to the reader that the relationship could also be expressed as *one-to-many*. However, since the convention in the E-R Diagram is to have the *one or more* cardinality on the left, it is usually expressed as M:1. All the samples in Example 7.4 are M:1 relationships. The *many-to-many relationship* is one where there is a degree of *one or more* in both directions. This relationship can be denoted as M:M (M to M) or M:N (M to N). Because the actual number of each degree is usually not the same, we will use the M:N notation. Fig. 7-5 shows an example of M:N relationship. Members can serve on *one or more* committee, and committees are made up of *one or more* members. Both M:1 and M:N relationships are very common. They are usually optional in both directions, but can be mandatory in one direction. It is unusual to find M:1 or M:N relationships that are mandatory in both directions.

Example 7.6
Examine the diagrams and identify the kinds of relationships illustrated.



a. Because a book can be written by one or more authors, and an author can write one or more books, this is an M:N relationship.
b. Because a client may be treated by one and only one therapist, and a therapist may be counselor of one or more clients, this is an M:1 relationship.

7.6 Normalizing the Model

Once the initial E-R Diagram has been formed, it must be normalized. The steps listed in Chapter 5 must be followed to put the model into 1NF, 2NF, 3NF or BCNF. We cannot overemphasize the importance of normalizing the model before the database is implemented using a DBMS. It is much harder to change the structure of the database after data has been entered into the tables.

Example 7.7
Refer to the PROJECT table in Fig. 5-2. That table could have been initially modeled as shown in Fig. 7-6. Is the entity in 1NF? If not, how can it be normalized?

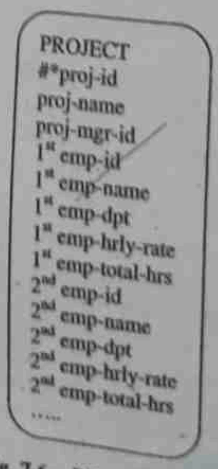


Fig. 7-6. PROJECT entity.

The objective of putting an entity into 1NF is to remove its repeating groups and ensure that all entries of the resulting table have at most a single value. It is clear in Fig. 7-6 that there are repeating values for each employee working on the project. Therefore, it is NOT in 1NF. Two entities are represented, PROJECT-EMPLOYEE and PROJECT. The first attempt at normalization from Chapter 5 Example 5.2 would result in the E-R Diagram in Fig. 7-7.

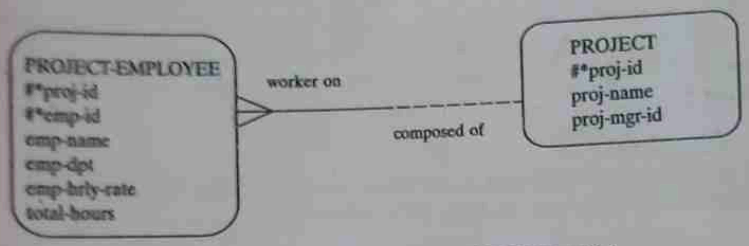


Fig. 7-7. PROJECT-EMPLOYEE and PROJECT entities.

Example 7.8
Put the model from Fig. 7-7 into 2NF.

Changing to 2NF assures that no nonprime attribute is partially dependent on any key. In this case, the only attribute that fully depends on the composite key (Proj-id,Emp-id) is Total-Hours. There would now be two entities, PROJECT-EMPLOYEE and HOURS-ASSIGNED, as shown in Fig. 7-8.

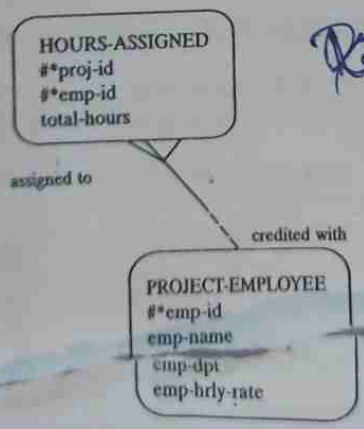


Fig. 7-8. HOURS-ASSIGNED and PROJECT-EMPLOYEE entities.

In the same way, an entity model can usually be changed into 3NF and BCNF using the normalization processes illustrated in Chapter 5. Performing these changes assures that the E-R Model and, therefore, the database, will be normalized from the beginning of its design.