# Preface

## (Fundamentals of Database Systems, Third Edition)





This book introduces the fundamental concepts necessary for designing, using, and implementing database systems and applications. Our presentation stresses the fundamentals of database modeling and design, the languages and facilities provided by database management systems, and system implementation techniques. The book is meant to be used as a textbook for a one- or two-semester course in database systems at the junior, senior, or graduate level, and as a reference book. We assume that readers are familiar with elementary programming and data-structuring concepts and that they have had some exposure to basic computer organization.

We start in Part 1 with an introduction and a presentation of the basic concepts from both ends of the database spectrum—conceptual modeling principles and physical file storage techniques. We conclude the book in Part 6 with an introduction to influential new database models, such as active, temporal, and deductive models, along with an overview of emerging technologies and applications, such as data mining, data warehousing, and Web databases. Along the way—in Part 2 through Part 5—we provide an indepth treatment of the most important aspects of database fundamentals.

The following key features are included in the third edition:

- The entire book has a self-contained, flexible organization that can be tailored to individual needs.
- Complete and updated coverage is provided on the relational model—including new material on Oracle and Microsoft Access as examples of relational systems—in Part 2.
- A comprehensive new introduction is provided on object databases and object-relational systems in Part 3, including the ODMG object model and the OQL query language, as well as an overview of object-relational features of SQL3, INFORMIX, and ORACLE 8.
- Updated coverage of EER conceptual modeling has been moved to Chapter 4 to follow the basic ER modeling in Chapter 3, and includes a new section on notation for UML class diagrams.
- Two examples running throughout the book—called COMPANY and UNIVERSITY—allow the reader to compare different approaches that use the same application.

- Coverage has been updated on database design, including conceptual design, normalization techniques, physical design, and database tuning.

# Part 1: Basic Concepts

## (Fundamentals of Database Systems, Third Edition)



**Chapter 1: Databases and Database Users**



**D**atabases and database systems have become an essential component of everyday life in modern society. In the course of a day, most of us encounter several activities that involve some interaction with a database. For example, if we go to the bank to deposit or withdraw funds; if we make a hotel or airline reservation; if we access a computerized library catalog to search for a bibliographic item; or if we order a magazine subscription from a publisher, chances are that our activities will involve someone accessing a database. Even purchasing items from a supermarket nowadays in many cases involves an automatic update of the database that keeps the inventory of supermarket items.

The above interactions are examples of what we may call **traditional database applications,** where most of the information that is stored and accessed is either textual or numeric. In the past few years, advances in technology have been leading to exciting new applications of database systems. **Multimedia databases** can now store pictures, video clips, and sound messages. **Geographic information systems (GIS)** can store and analyze maps, weather data, and satellite images. **Data warehouses** and **on-line analytical processing (OLAP)** systems are used in many companies to extract and analyze useful information from very large databases for decision making. **Real-time** and **active database technology** is used in controlling industrial and manufacturing processes. And database search techniques are being applied to the World Wide Web to improve the search for information that is needed by users browsing through the Internet.

To understand the fundamentals of database technology, however, we must start from the basics of traditional database applications. So, in Section 1.1 of this chapter we define what a database is, and then we give definitions of other basic terms. In Section 1.2, we provide a simple UNIVERSITY database example to illustrate our discussion. Section 1.3 describes some of the main characteristics of database systems, and Section 1.4 and Section 1.5 categorize the types of personnel whose jobs involve using and interacting with database systems. Section 1.6, Section 1.7, and Section 1.8 offer a more thorough discussion of the various capabilities provided by database systems and of the implications of using the database approach. Section 1.9 summarizes the chapter.

The reader who desires only a quick introduction to database systems can study Section 1.1 through Section 1.5, then skip or browse through Section 1.6, Section 1.7 and Section 1.8 and go on to Chapter 2.

## 1.1 Introduction

Databases and database technology are having a major impact on the growing use of computers. It is fair to say that databases play a critical role in almost all areas where computers are used, including business, engineering, medicine, law, education, and library science, to name a few. The word *database* is in such common use that we must begin by defining a database. Our initial definition is quite general.

A **database** is a collection of related data (Note 1). By **data,** we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. You may have recorded this data in an indexed address book, or you may have stored it on a diskette, using a personal computer and software such as DBASE IV or V, Microsoft ACCESS, or EXCEL. This is a collection of related data with an implicit meaning and hence is a database.

The preceding definition of *database* is quite general; for example, we may consider the collection of words that make up this page of text to be related data and hence to constitute a database. However, the common use of the term *database* is usually more restricted. A database has the following implicit properties:

- A database represents some aspect of the real world, sometimes called the **miniworld** or the **Universe of Discourse (UoD).** Changes to the miniworld are reflected in the database.
- A database is a logically coherent collection of data with some inherent meaning. A random assortment of data cannot correctly be referred to as a database.
- A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.

In other words, a database has some source from which data are derived, some degree of interaction with events in the real world, and an audience that is actively interested in the contents of the database.

A database can be of any size and of varying complexity. For ex
ample, the list of names and addresses referred to earlier may consist of only a few hundred records, each with a simple structure. On the other hand, the card catalog of a large library may contain half a million cards stored under different categories—by primary author's last name, by subject, by book title—with each category organized in alphabetic order. A database of even greater size and complexity is maintained by the Internal Revenue Service to keep track of the tax forms filed by U.S. taxpayers. If we assume that there are 100 million tax-payers and if each taxpayer files an average of five forms with approximately 200 characters of information per form, we would get a database of 100*(10
6
)*200*5 characters (bytes) of information. If the IRS keeps the past three returns for each taxpayer in addition to the current return, we would get a database of 4*(10
1
) bytes (400 gigabytes). This huge amount of information must be organized and managed so that users can search for, retrieve, and update the data as needed.

A database may be generated and maintained manually or it may be computerized. The library card catalog is an example of a database that may be created and maintained manually. A computerized database may be created and maintained either by a group of application programs written specifically for that task or by a database management system.

A **database management system (DBMS)** is a collection of programs that enables users to create and maintain a database. The DBMS is hence a *general-purpose software system* that facilitates the processes of *defining*, *constructing*, and *manipulating* databases for various applications. **Defining** a database involves specifying the data types, structures, and constraints for the data to be stored in the database. **Constructing** the database is the process of storing the data itself on some storage medium that is controlled by the DBMS. **Manipulating** a database includes such functions as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data.

It is not necessary to use general-purpose DBMS software to implement a computerized database. We could write our own set of programs to create and maintain the database, in effect creating our own *special-purpose* DBMS software. In either case—whether we use a general-purpose DBMS or not—we usually have to employ a considerable amount of software to manipulate the database. We will call the database and DBMS software together a **database system.** Figure 01.01 illustrates these ideas.

## 1.2 An Example

Let us consider an example that most readers may be familiar with: a UNIVERSITY database for maintaining information concerning students, courses, and grades in a university environment. Figure 01.02 shows the database structure and a few sample data for such a database. The database is organized as five files, each of which stores data records of the same type (Note 2). The STUDENT file stores data on each student; the COURSE file stores data on each course; the SECTION file stores data on each section of a course; the GRADE_REPORT file stores the grades that students receive in the various sections they have completed; and the PREREQUISITE file stores the prerequisites of each course.

To *define* this database, we must specify the structure of the records of each file by specifying the different types of **data elements** to be stored in each record. In Figure 01.02, each STUDENT record includes data to represent the student's Name, StudentNumber, Class (freshman or 1, sophomore or 2, . . .), and Major (MATH, computer science or CS, . . .); each COURSE record includes data to represent the CourseName, CourseNumber, CreditHours, and Department (the department that offers the course); and so on. We must also specify a **data type** for each data element within a record. For example, we can specify that Name of STUDENT is a string of alphabetic characters, StudentNumber of STUDENT is an integer, and Grade of GRADE_REPORT is a single character from the set {A, B, C, D, F, I}. We may also use a coding scheme to represent a data item. For example, in Figure 01.02 we represent the Class of a STUDENT as 1 for freshman, 2 for sophomore, 3 for junior, 4 for senior, and 5 for graduate student.

To *construct* the UNIVERSITY database, we store data to represent each student, course, section, grade report, and prerequisite as a record in the appropriate file. Notice that records in the various files may

be related. For example, the record for "Smith" in the STUDENT file is related to two records in the GRADE_REPORT file that specify Smith's grades in two sections. Similarly, each record in the

PREREQUISITE file relates two course records: one representing the course and the other representing the prerequisite. Most medium-size and large databases include many types of records and have

many relationships among the records.

Database *manipulation* involves querying and updating. Examples of queries are "retrieve the transcript—a list of all courses and grades—of Smith"; "list the names of students who took the section of the Database course offered in fall 1999 and their grades in that section"; and "what are the prerequisites of the Database course?" Examples of updates are "change the class of Smith to Sophomore"; "create a new section for the Database course for this semester"; and "enter a grade of A for Smith in the Database section of last semester." These informal queries and updates must be specified precisely in the database system language before they can be processed.

## 1.3 Characteristics of the Database Approach

1.3.1 Self-Describing Nature of a Database System
1.3.2 Insulation between Programs and Data, and Data Abstraction
1.3.3 Support of Multiple Views of the Data
1.3.4 Sharing of Data and Multiuser Transaction Processing

A number of characteristics distinguish the database approach from the traditional approach of programming with files. In traditional **file processing,** each user defines and implements the files needed for a specific application as part of programming the application. For example, one user, the *grade reporting office*, may keep a file on students and their grades. Programs to print a student's transcript and to enter new grades into the file are implemented. A second user, the accounting office, may keep track of students' fees and their payments. Although both users are interested in data about students, each user maintains separate files—and programs to manipulate these files—because each requires some data not available from the other user's files. This redundancy in defining and storing data results in wasted storage space and in redundant efforts to maintain common data up-to-date.

In the database approach, a single repository of data is maintained that is defined once and then is accessed by various users. The main characteristics of the database approach versus the file-processing approach are the following.

### 1.3.1 Self-Describing Nature of a Database System

A fundamental characteristic of the database approach is that the database system contains not only the database itself but also a complete definition or description of the database structure and constraints. This definition is stored in the system **catalog,** which contains information such as the structure of each file, the type and storage format of each data item, and various constraints on the data. The information stored in the catalog is called **meta-data,** and it describes the structure of the primary database (Figure 01.01).

The catalog is used by the DBMS software and also by database users who need information about the database structure. A general purpose DBMS software package is not written for a specific database application, and hence it must refer to the catalog to know the structure of the files in a specific database, such as the type and format of data it will access. The DBMS software must work equally well with *any number of database applications*—for example, a university database, a banking database, or a company database—as long as the database definition is stored in the catalog.

In traditional file processing, data definition is typically part of the application programs themselves. Hence, these programs are constrained to work with only *one specific database*, whose structure is declared in the application programs. For example, a PASCAL program may have record structures declared in it; a C++ program may have "struct" or "class" declarations; and a COBOL program has Data Division statements to define its files. Whereas file-processing software can access only specific databases, DBMS software can access diverse databases by extracting the database definitions from the catalog and then using these definitions.

In the example shown in Figure 01.02, the DBMS stores in the catalog the definitions of all the files shown. Whenever a request is made to access, say, the Name of a STUDENT record, the DBMS software refers to the catalog to determine the structure of the STUDENT file and the position and size of the Name data item within a STUDENT record. By contrast, in a typical file-processing application, the file structure and, in the extreme case, the exact location of Name within a STUDENT record are already coded within each program that accesses this data item.

### 1.3.2 Insulation between Programs and Data, and Data Abstraction

In traditional file processing, the structure of data files is embedded in the access programs, so any changes to the structure of a file may require *changing all programs* that access this file. By contrast, DBMS access programs do not require such changes in most cases. The structure of data files is stored in the DBMS catalog separately from the access programs. We call this property **program-data independence.** For example, a file access program may be written in such a way that it can access only STUDENT records of the structure shown in Figure 01.03. If we want to add another piece of data to each STUDENT record, say the Birthdate, such a program will no longer work and must be changed. By contrast, in a DBMS environment, we just need to change the description of STUDENT records in the catalog to reflect the inclusion of the new data item Birthdate; no programs are changed. The next time a DBMS program refers to the catalog, the new structure of STUDENT records will be accessed and used.

In object-oriented and object-relational databases (see Part III), users can define operations on data as part of the database definitions. An **operation** (also called a *function*) is specified in two parts. The *interface* (or *signature*) of an operation includes the operation name and the data types of its arguments (or parameters). The *implementation* (or *method*) of the operation is specified separately and can be changed without affecting the interface. User application programs can operate on the data by invoking these operations through their names and arguments, regardless of how the operations are implemented. This may be termed **program-operation independence.**

The characteristic that allows program-data independence and program-operation independence is called **data abstraction.** A DBMS provides users with a **conceptual representation** of data that does not include many of the details of how the data is stored or how the operations are implemented. Informally, a **data model** is a type of data abstraction that is used to provide this conceptual representation. The data model uses logical concepts, such as objects, their properties, and their interrelationships, that may be easier for most users to understand than computer storage concepts. Hence, the data model *hides* storage and implementation details that are not of interest to most database users.

For example, consider again Figure 01.02. The internal implementation of a file may be defined by its record length—the number of characters (bytes) in each record—and each data item may be specified

by its starting byte within a record and its length in bytes. The STUDENT

record would thus be represented as shown in Figure 01.03. But a typical database user is not concerned with the location of each data item within a record or its length; rather the concern is that, when a reference is made to Name of STUDENT, the correct value is returned. A conceptual representation of the STUDENT records is shown in Figure 01.02.

Many other details of file-storage organization—such as the access paths specified on a file—can be hidden from database users by the DBMS;

we will discuss storage details in Chapter 5 and Chapter 6.

In the database approach, the detailed structure and organization of each file are stored in the catalog. Database users refer to the conceptual representation of the files, and the DBMS extracts the details of file storage from the catalog when these are needed by the DBMS software. Many data models can be used to provide this data abstraction to database users. A major part of this book is devoted to presenting various data models and the concepts they use to abstract the representation of data.

With the recent trend toward object-oriented and object-relational databases, abstraction is carried one level further to include not only the data structure but also the operations on the data. These operations provide an abstraction of miniworld activities commonly understood by the users. For example, an operation CALCULATE_GPA can be applied to a student object to calculate the grade point average. Such operations can be invoked by the user queries or programs without the user knowing the details of how they are internally implemented. In that sense, an abstraction of the miniworld activity is made available to the user as an **abstract operation.**

### 1.3.3 Support of Multiple Views of the Data

A database typically has many users, each of whom may require a different perspective or **view** of the database. A view may be a subset of the database or it may contain **virtual data** that is derived from the database files but is not explicitly stored. Some users may not need to be aware of whether the data they refer to is stored or derived. A multiuser DBMS whose users have a variety of applications must provide facilities for defining multiple views. For example, one user of the database of Figure 01.02 may be interested only in the transcript of each student; the view for this user is shown in Figure 01.04(a). A second user, who is interested only in checking that students have taken all the prerequisites of each course they register for, may require the view shown in Figure 01.04(b).

### 1.3.4 Sharing of Data and Multiuser Transaction Processing

A multiuser DBMS, as its name implies, must allow multiple users to access the database at the same time. This is essential if data for multiple applications is to be integrated and maintained in a single database. The DBMS must include **concurrency control** software to ensure that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct. For example, when several reservation clerks try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one clerk at a time for assignment to a passenger. These types of applications are generally called **on-line transaction processing (OLTP)** applications. A fundamental role of multiuser DBMS software is to ensure that concurrent transactions operate correctly.

The preceding characteristics are most important in distinguishing a DBMS from traditional file-processing software. In Section 1.6 we discuss additional functions that characterize a DBMS. First, however, we categorize the different types of persons who work in a database environment

# DATABASE

## SYSTEM CONCEPTS

S I XTH E D I T I ON

Abraham Silberschatz
*Yale University*
Henry F. Korth
*Lehigh University*
S. Sudarshan
*Indian Institute of Technology, Bombay*

• **Data redundancy and inconsistency**. Since different programmers create the files and application programs over a long period, the various files are likely to have different structures and the programs may bewritten in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, if a student has a double major (say, music and mathematics) the address and telephone number of that student may appear in a file that consists of student records of students in the Music department and in a file that consists of student records of students in the Mathematics department. This redundancy leads to higher storage and access cost. In addition, it may lead to **data inconsistency**; that is, the various copies of the same datamayno longer agree. For example, a changed student address may be reflected in the Music department records but not elsewhere in the system.

• **Difficulty in accessing data**. Suppose that one of the university clerks needs to find out the names of all students who live within a particular postal-code area. The clerk asks the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of *all* students. The university clerk has now two choices: either obtain the list of all students and extract the needed information manually or ask a programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same clerk needs to trim that list to include only those students who have taken at least 60 credit hours. As expected, a program to generate such a list does not exist. Again, the clerk has the preceding two options, neither of which is satisfactory.

The point here is that conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficientmanner. More responsive data-retrieval systems are required for general use.

• **Data isolation**. Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

• **Integrity problems**. The data values stored in the database must satisfy certain types of **consistency constraints**. Suppose the university maintains an account for each department, and records the balance amount in each account. Suppose also that the university requires that the account balance of a department may never fall below zero. Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compoundedwhen constraints involve several data items from different files.

• **Atomicity problems**. A computer system, like any other device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data

**5**

be restored to the consistent state that existed prior to the failure. Consider a program to transfer $500 from the account balance of department *A* to the account balance of department *B*. If a system failure occurs during the execution of the program, it is possible that the $500 was removed from the balance of department *A* butwas not credited to the balance of department *B*, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur. That is, the funds transfer must be *atomic*—it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

• **Concurrent-access anomalies**. For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. Indeed, today, the largest Internet retailers may have millions of accesses per day to their data by shoppers. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data. Consider department *A*, with an account balance of $10,000. If two department clerks debit the account balance (by say $500 and $100, respectively) of department *A* at almost exactly the same time, the result of the concurrent executions may leave the budget in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount beingwithdrawn, and write the result back. If the two programs run concurrently, they may both read the value $10,000, and write back $9500 and $9900, respectively. Depending on which one writes the value last, the account balance of department *A* may contain either $9500 or $9900, rather than the correct value of $9400. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously.
As another example, suppose a registration program maintains a count of students registered for a course, in order to enforce limits on the number of students registered.When a student registers, the program reads the current count for the courses, verifies that the count is not already at the limit, adds one to the count, and stores the count back in the database. Suppose two students register concurrently, with the count at (say) 39. The two program executions may both read the value 39, and both would then write back 40, leading to an incorrect increase of only 1, even though two students successfully registered for the course and the count should be 41. Furthermore, suppose the course registration limit was 40; in the above case both students would be able to register, leading to a violation of the limit of 40 students.

• **Security problems**. Not every user of the database system should be able to access all the data. For example, in a university, payroll personnel need to see only that part of the database that has financial information. They do not need access to information about academic records. But, since application programs are added to the file-processing system in an ad hoc manner, enforcing such security constraints is difficult.

These difficulties, among others, prompted the development of database systems. In what follows, we shall see the concepts and algorithms that enable database systems to solve the problems with file-processing systems. In most of this book, we use a university organization as a running example of a typical data-processing application.

## 1.3 View of Data

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data. A major purpose of a database system is to provide users with an *abstract* view of the data. That is, the system hides certain details of how the data are stored and maintained.

### 1.3.1 Data Abstraction

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led designers to use complex data structures to represent data in the database. Since many database-system users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system:

• **Physical level**. The lowest level of abstraction describes *how* the data are actually stored. The physical level describes complex low-level data structures in detail.

• **Logical level**. The next-higher level of abstraction describes *what* data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as **physical data independence**. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.

• **View level**. The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

Figure 1.1 shows the relationship among the three levels of abstraction. An analogy to the concept of data types in programming languages may clarify the distinction among levels of abstraction. Many high-level programming

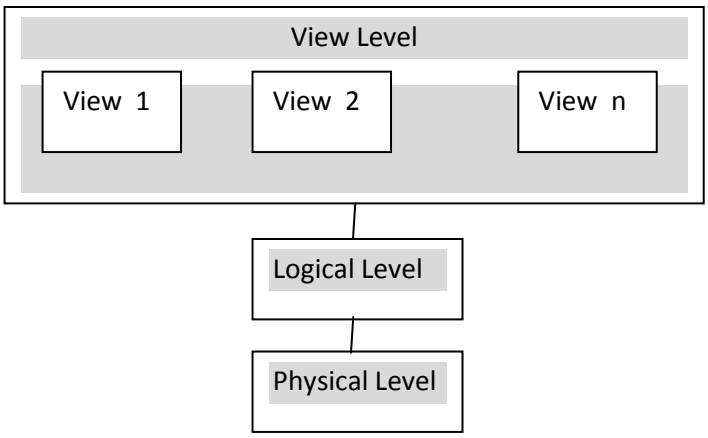



**Figure 1.1** The three levels of data abstraction.

languages support the notion of a structured type. For example, we may describe a record as follows:1

**type** *instructor* = **record**
*ID* : **char** (5);
*name* : **char** (20);
*dept name* : **char** (20);
*salary* : **numeric** (8,2);
**end**;

This code defines a new record type called *instructor* with four fields. Each field has a name and a type associated with it. A university organization may have several such record types, including

• *department*, with fields *dept name*, *building*, and *budget*
• *course*, with fields *course id*, *title*, *dept name*, and *credits*
• *student*, with fields *ID*, *name*, *dept name*, and *tot cred*

At the physical level, an *instructor*, *department*, or *student* record can be described as a block of consecutive storage locations. The compiler hides this level of detail from programmers. Similarly, the database system hides many of the lowest-level storage details from database programmers. Database administrators, on the other hand, may be aware of certain details of the physical organization of the data.

1The actual type declaration depends on the language being used. C and C++ use **struct** declarations. Java does not have
such a declaration, but a simple class can be defined to the same effect.

**8**

At the logical level, each such record is described by a type definition, as in the previous code segment, and the interrelationship of these record types is defined as well. Programmers using a programming language work at this level of abstraction. Similarly, database administrators usually work at this level of abstraction.

Finally, at the view level, computer users see a set of application programs that hide details of the data types. At the view level, several views of the database are defined, and a database user sees some or all of these views. In addition to hiding details of the logical level of the database, the views also provide a security mechanism to prevent users from accessing certain parts of the database. For example, clerks in the university registrar office can see only that part of the database that has information about students; they cannot access information about salaries of instructors.

### 1.3.2 Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**. Schemas are changed infrequently, if at all.

The concept of database schemas and instances can be understood by analogy to a program written in a programming language.Adatabase schema corresponds to the variable declarations (along with associated type definitions) in a program. Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema.

Database systems have several schemas, partitioned according to the levels of abstraction. The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called **subschemas**, that describe different views of the database.

Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit **physical data independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

We study languages for describing schemas after introducing the notion of data models in the next section.

### 1.3.3 Data Models

Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. A data model provides a way to describe the design of a database at the physical, logical, and view levels.

**9**

There are a number of different data models that we shall cover in the text.
The data models can be classified into four different categories:

• **Relational Model**. The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as **relations**. The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes.
The columns of the table correspond to the attributes of the record type. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.

Chapters 2 through 8 cover the relational model in detail.

• **Entity-Relationship Model**. The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and*relationships* among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects. The entity-relationship model is widely used in database design, and Chapter 7 explores it in detail.
• **Object-Based Data Model**.Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led to the development of an object-oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. The object-relational data model combines features of the object-oriented data model and relational data model. Chapter 22 examines the object-relational data model.
• **Semistructured Data Model**. The semistructured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The **Extensible Markup Language (XML)** is widely used to represent semistructured data. Chapter 23 covers it.
Historically, the **network data model** and the **hierarchical data model** preceded the relational data model. These modelswere tied closely to the underlying implementation, and complicated the task of modeling data. As a result they are used little now, except in old database code that is still in service in some places. They are outlined online in Appendices D and E for interested readers.

**1.4 Database Languages**

A database system provides a **data-definition language** to specify the database schema and a **data-manipulation language** to express database queries and up

information is updated without taking precautions to update all copies of the information. For example, different offerings of a course may have the same course identifier, but may have different titles. It would then become unclear what the correct title of the course is. Ideally, information should appear in exactly one place.
2. Incompleteness: A bad design may make certain aspects of the enterprise difficult or impossible to model. For example, suppose that, as in case (1) above, we only had entities corresponding to course offering, without having an entity corresponding to courses. Equivalently, in terms of relations, suppose we have a single relation where we repeat all of the course information once for each section that the course is offered. It would then be impossible to represent information about a new course, unless that course is offered. We might try to make do with the problematic design by storing null values for the section information. Such a work-around is not only unattractive, but may be prevented by primary-key constraints.
Avoiding bad designs is not enough. There may be a large number of good designs from which we must choose. As a simple example, consider a customer whobuys a product. Is the sale of this product a relationship between the customer and the product? Alternatively, is the sale itself an entity that is related both to the customer and to the product? This choice, though simple, may make an important difference in what aspects of the enterprise can bemodeledwell. Considering the need to make choices such as this for the large number of entities and relationships in a real-world enterprise, it is not hard to see that database design can be a challenging problem. Indeed we shall see that it requires a combination of both science and "good taste."

## 7.2 The Entity-Relationship Model

The entity-relationship (E-R) data model was developed to facilitate database design by allowing specification of an *enterprise schema* that represents the overall logical structure of a database.
The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. Because of this usefulness, many database-design tools draw on concepts from the E-R model. The E-R data model employs three basic concepts: entity sets, relationship sets, and attributes, which we study first. The E-R model also has an associated diagrammatic representation,
the E-R diagram, which we study later in this chapter.

### 7.2.1 Entity Sets

An entity is a "thing" or "object" in the real world that is distinguishable from all other objects. For example, each person in a university is an entity. An entity has a set of properties, and the values for some set of properties may uniquely identify an entity. For instance, a person may have a *person id* property whose

value uniquely identifies that person. Thus, the value 677-89-9011 for *person id* would uniquely identify one particular person in the university. Similarly, courses can be thought of as entities, and *course id* uniquely identifies a course entity in the university. An entity may be concrete, such as a person or a book, or it may be abstract, such as a course, a course offering, or a flight reservation.

An entity set is a set of entities of the same type that share the same properties, or attributes. The set of all people who are instructors at a given university, for example, can be defined as the entity set *instructor*. Similarly, the entity set *student* might represent the set of all students in the university.

In the process of modeling, we often use the term *entity set* in the abstract, without referring to a particular set of individual entities. We use the term extension of the entity set to refer to the actual collection of entities belonging to the entity set. Thus, the set of actual instructors in the university forms the extension of the entity set *instructor*. The above distinction is similar to the difference between a relation and a relation instance, which we saw in Chapter 2.

Entity sets do not need to be disjoint. For example, it is possible to define the entity set of all people in a university (*person*). A *person* entity may be an *instructor* entity, a *student* entity, both, or neither.

An entity is represented by a set of attributes. Attributes are descriptive properties possessed by each member of an entity set. The designation of an attribute for an entity set expresses that the database stores similar information concerning each entity in the entity set; however, each entity may have its own value for each attribute. Possible attributes of the *instructor* entity set are *ID*, *name*, *dept name*, and *salary*. In real life, there would be further attributes, such as street number, apartment number, state, postal code, and country, but we omit them to keep our examples simple. Possible attributes of the *course* entity set are *course id*, *title*, *dept name*, and *credits*.

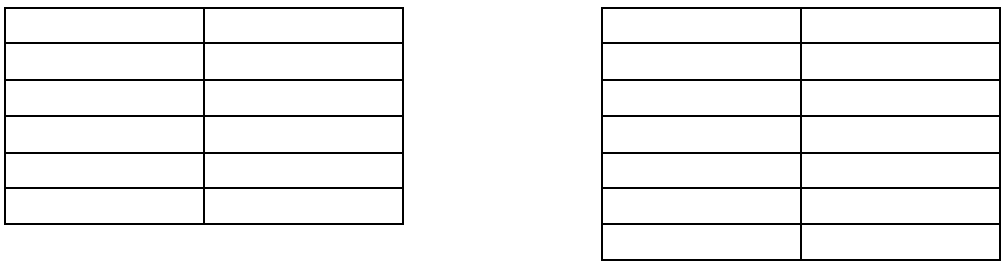
Each entity has a value for each of its attributes. For instance, a particular *instructor* entity may have the value 12121 for *ID*, the valueWu for *name*, the value Finance for *dept name*, and the value 90000 for *salary*.

The *ID* attribute is used to identify instructors uniquely, since there may be more than one instructor with the same name. In the United States, many enterprises find it convenient to use the *social-security* number of a person2 as an attribute whose value uniquely identifies the person. In general the enterprise would have to create and assign a unique identifier for each instructor.

A database thus includes a collection of entity sets, each of which contains any number of entities of the same type. Figure 7.1 shows part of a university database that consists of two entity sets: *instructor* and *student*. To keep the figure simple, only some of the attributes of the two entity sets are shown.

A database for a university may include a number of other entity sets. For example, in addition to keeping track of instructors and students, the university also has information about courses, which are represented by the entity set *course*

2In the United States, the government assigns to each person in the country a unique number, called a social-security number, to identify that person uniquely. Each person is supposed to have only one social-security number, and no two people are supposed to have the same social-security number.



Instructor student

Figure 7.1 Entity sets *instructor* and *student*.

with attributes *course id*, *title*, *dept name* and *credits*. In a real setting, a university database may keep dozens of entity sets.

## 7.2.2 Relationship Sets

A relationship is an association among several entities.

For example, we can
define a relationship *advisor* that associates instructor Katz with student Shankar.
This relationship specifies that Katz is an advisor to student Shankar.

A relationship set is a set of relationships of the same type.

Formally, it is a
mathematical relation on $n \geq 2$ (possibly nondistinct) entity sets. If $E1, E2, \ldots, En$ are entity sets, then a relationship set $R$ is a subset of
$\{(e1, e2, \ldots, en) \mid e1 \in E1, e2 \in E2, \ldots, en \in En\}$
where $(e1, e2, \ldots, en)$ is a relationship.
Consider the two entity sets *instructor* and *student* in Figure 7.1. We define the relationship set *advisor* to denote the association between instructors and students. Figure 7.2 depicts this association.
As another example, consider the two entity sets *student* and *section*. We can define the relationship set *takes* to denote the association between a student and the course sections in which that student is enrolled.
The association between entity sets is referred to as participation; that is, the entity sets $E1, E2, \ldots, En$ participate in relationship set $R$. A relationship instance in an E-R schema represents an association between the named entities in the real-world enterprise that is being modeled. As an illustration, the individual *instructor* entity Katz,who has instructor *ID* 45565, and the *student* entity Shankar, who has student *ID* 12345, participate in a relationship instance of *advisor*. This relationship instance represents that in the university, the instructor Katz is advising student Shankar.
The function that an entity plays in a relationship is called that entity's role.
Since entity sets participating in a relationship set are generally distinct, roles

# ER Model Relationships in general

Relationship: Interaction between entities

Indicator : an attribute of one entity refers to another entity (Represent such references as

relationships not attributes)

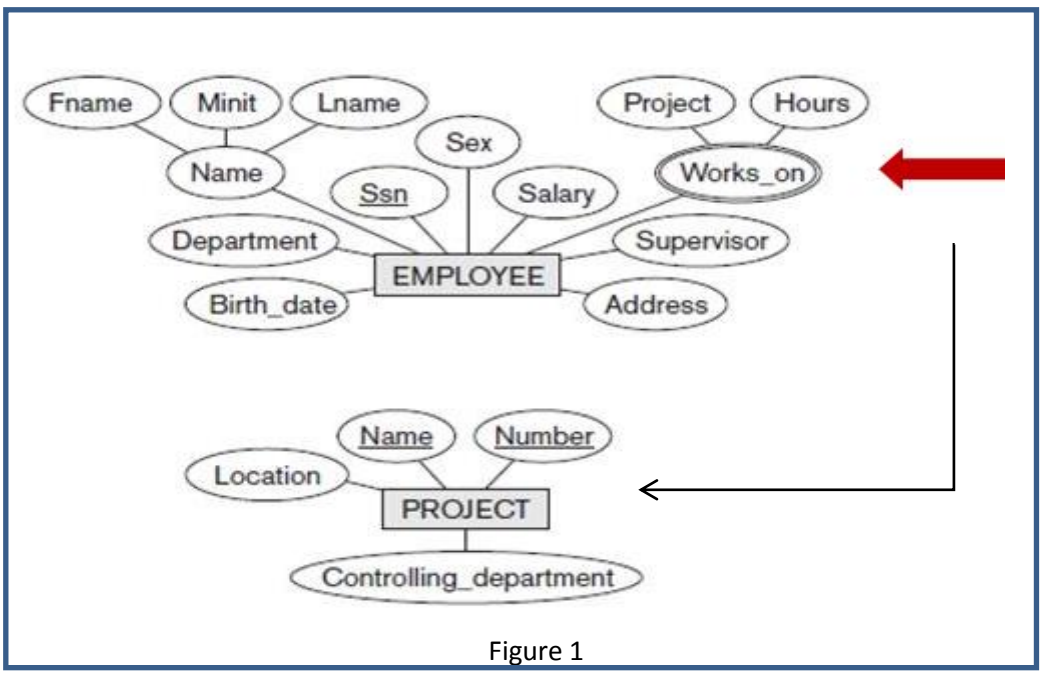



Figure 1

Diagramming Relationship : Diamond for relationship type , connected to each participating entity type

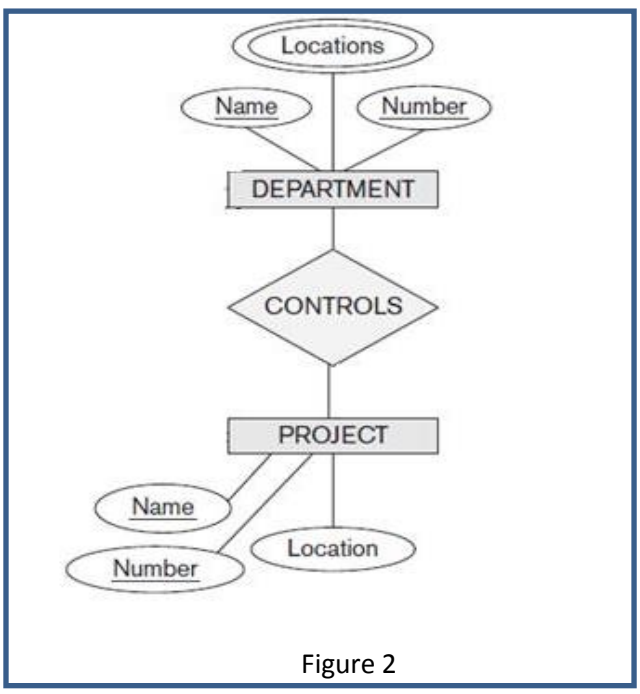
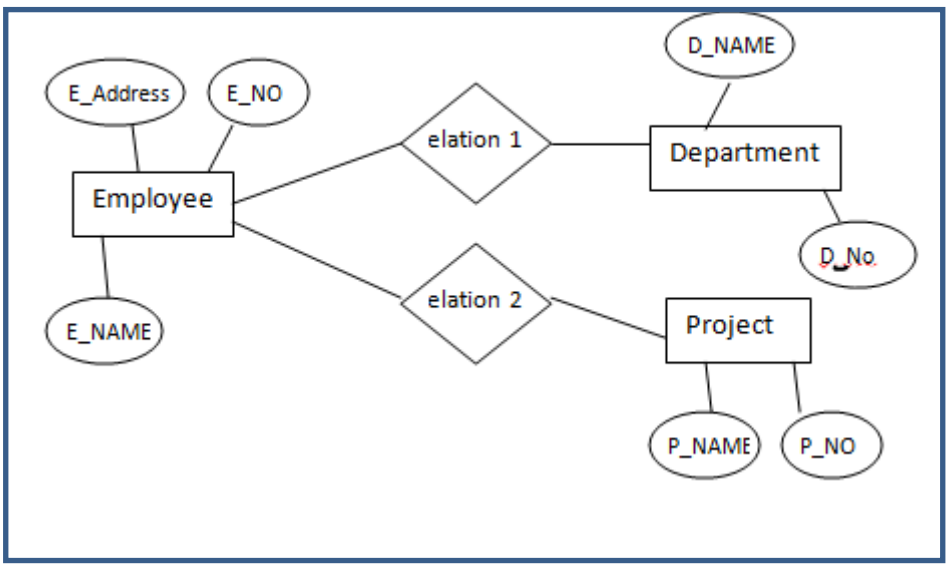



Figure 2

Example 1 : In a company, there are three entity :

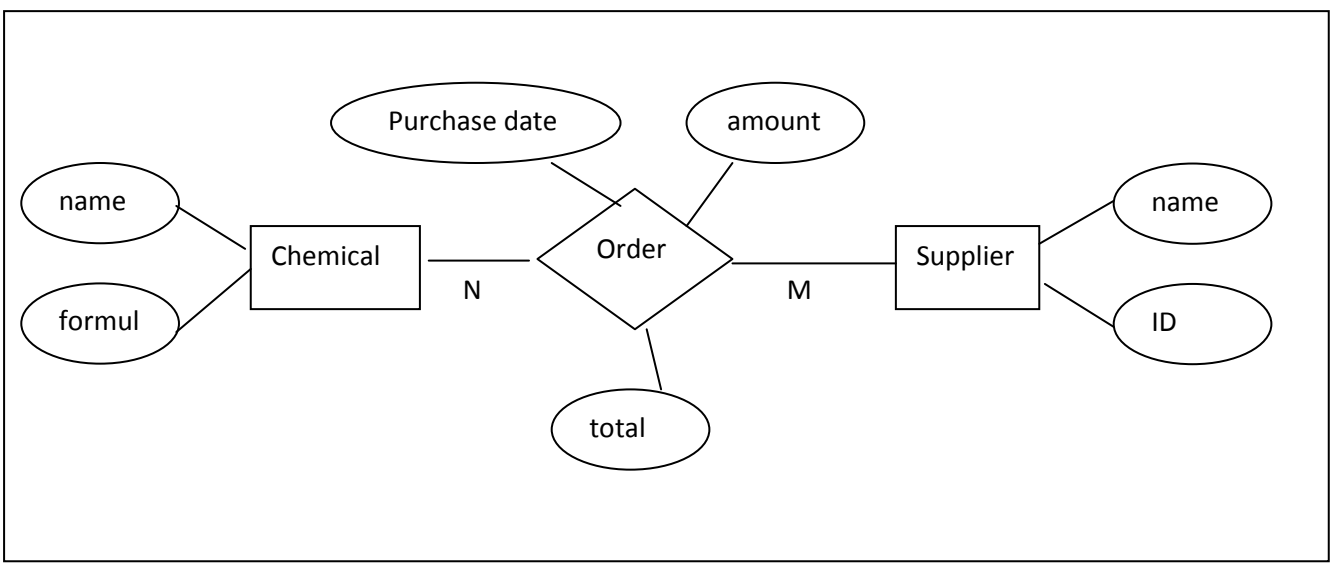
1- Employees : E_NO ,E_NAME, E_ADDRESS
2- Departments : D_NO ,D_NAME
3-Projects : P_NAME , P_NO

Each employee belongs to a department and each project has many employees works on it.
There are two relations that connect the three entities




Example 2 : A chemical factory producing chemical materials , each material identified by a name and a formula .
The supplier , identified by his name and his ID , purchase from the factory by an order . The order has date , amount and total.

To draw the ER model
Each supplier can have any material , and any material can go to any supplier .
The relation is of type **many-to-many**.



# Cardinality

http://ion.uwinnipeg.ca/~rmcfadye/2914/relationships.pdf(ACS 2914 ERD: Relationships Ron McFadyen)

Cardinality is a constraint on a relationship specifying the number of entity instances that a specific entity may be related to via the relationship. Consider the relationship "works in".


When we ask How many employees can work in a single department? or How many departments can an employee work in? we are asking questions regarding the cardinality of the relationship. The three classifications are: one-to-one, one-to-many, and many-to-many. Below, the ERD shows a relationship between invoice lines and products.


The "n" represents an "arbitrary number of instances", and the "1" represents "at most one instance". We interpret the cardinality specifications with the following business rule statements:

- The "n" indicates that the same Product entity can be specified on "any number of" Invoice Lines.
- The "1" indicates that an Invoice Line entity specifies "at most one" Product entity

Exercise: Consider the University and entity types: Class, Course, Department, Student. What relationship types exist and what cardinalities apply to the various relationships and entities?

**One-to-One Relationships**

One-to-one relationships have 1 specified for both cardinalities, and do not seem to arise very often. To illustrate a one-to-one, we require very specific business rules.

Suppose we have People and Vehicles. Assume that we are only concerned with the current driver of a vehicle, and that we are only concerned with the current vehicle that a driver is operating. Then, we have a one-to-one relationship between Vehicle and Person (note the role shown for Person in this relationship):

**One-to-Many Relationships**

This type of relationship has 1 and n specified for cardinalities, and is very common in database designs. Suppose we have customers and orders and the business rules:

- An order is related to one customer, and
- A customer can have any number (zero or more) of orders.

We say there is a one-to-many relationship between customer and order, and we draw this as:


**Many-to-Many Relationships**

Many-to-many relationships have "many" specified for both cardinalities, and are also very common. However, should you examine a data model in some business, there is a good chance you will not see any many-to-many relationships on the diagram. In those cases, the data modeler has resolved the many-to-many relationships into two one-tomany relationships. Suppose we are interested in courses and students and the fact that students register for courses: Any student may take several courses, A course may be taken by several students. This situation is represented with a many-to-many relationship between Course and Student:

# ER Model to Relational Model

https://www.tutorialspoint.com/dbms/er_model_to_relational_model.htm

ER Model, when conceptualized into diagrams, gives a good overview of entity-relationship, which is easier to understand. ER diagrams can be mapped to relational schema, that is, it is possible to create relational schema using ER diagram. We cannot import all the ER constraints into relational model, but an approximate schema can be generated.

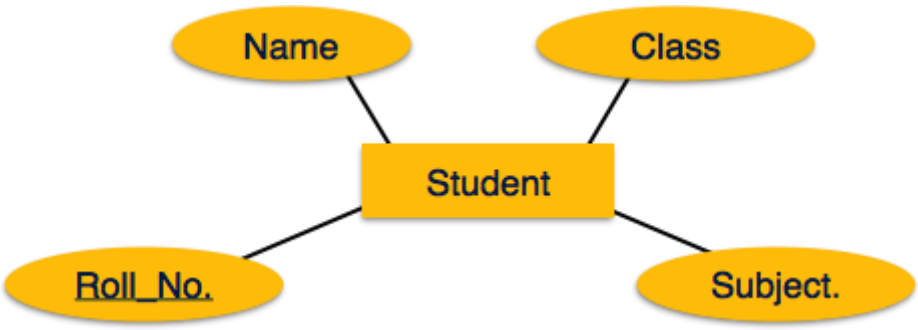
There are several processes and algorithms available to convert ER Diagrams into Relational Schema. Some of them are automated and some of them are manual. We may focus here on the mapping diagram contents to relational basics.

ER diagrams mainly comprise of –

- Entity and its attributes
- Relationship, which is association among entities.

## Mapping Entity

An entity is a real-world object with some attributes.

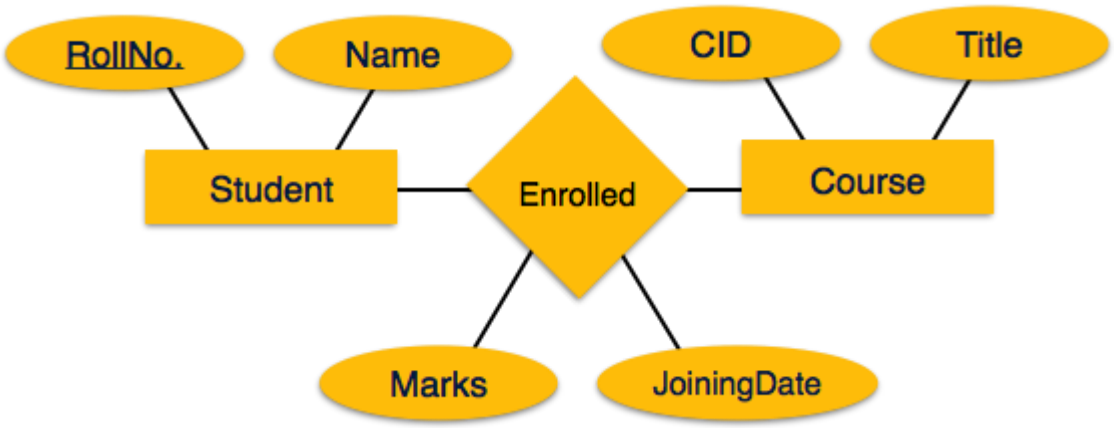



## Mapping Process

- Create table for each entity.
- Entity's attributes should become fields of tables with their respective data types.
- Declare primary key.

## Mapping Relationship

A relationship is an association among entities.




# Mapping Process

- Create table for a relationship.
- Add the primary keys of all participating Entities as fields of table with their respective data types.
- If relationship has any attribute, add each attribute as field of table.
- Declare a primary key composing all the primary keys of participating entities.
- Declare all foreign key constraints.

• *student* (*ID, name, dept name, tot cred*)
• *advisor* (*s id, i id*)
• *takes* (*ID, course id, sec id, semester, year, grade*)
• *classroom* (*building, room number, capacity*)
• *time slot* (*time slot id, day, start time, end time*)

## 2.3 Keys

We must have a way to specify how tuples within a given relation are distinguished. This is expressed in terms of their attributes. That is, the values of the attribute values of a tuple must be such that they can *uniquely identify* the tuple. In other words, no two tuples in a relation are allowed to have exactly the same value for all attributes.

A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation. For example, the *ID* attribute of the relation *instructor* is sufficient to distinguish one instructor tuple from another.
Thus, *ID* is a superkey. The *name* attribute of *instructor*, on the other hand, is not a superkey, because several instructors might have the same name.
Formally, let *R* denote the set of attributes in the schema of relation *r*. If we say that a subset *K* of *R* is a *superkey* for *r* , we are restricting consideration to instances of relations *r* in which no two distinct tuples have the same values on all attributes in *K*. That is, if *t*1 and *t*2 are in *r* and *t*1 _= *t*2, then *t*1.*K* _= *t*2.*K*.
A superkey may contain extraneous attributes. For example, the combination of *ID* and *name* is a superkey for the relation *instructor*. If *K* is a superkey, then so is any superset of *K*. We are often interested in superkeys for which no proper subset is a superkey. Such minimal superkeys are called **candidate keys**.
It is possible that several distinct sets of attributes could serve as a candidate key. Suppose that a combination of *name* and *dept name* is sufficient to distinguish among members of the *instructor* relation. Then, both *{ID}* and *{name, dept name}* are candidate keys. Although the attributes *ID* and *name* together can distinguish *instructor* tuples, their combination, *{ID, name}*, does not form a candidate key, since the attribute *ID* alone is a candidate key.

We shall use the term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation.

A key (whether primary, candidate, or super) is a property of the entire relation, rather than of the individual tuples.

Any two individual tuples in the relation are prohibited from having the same value on the key attributes at the same time.

The designation of a key represents a constraint in the real-world enterprise being modeled.
Primary keys must be chosen with care. As we noted, the name of a person is obviously not sufficient, because there may be many people with the same name.
In the United States, the social-security number attribute of a person would be a candidate key. Since non-U.S. residents usually do not have social-security

numbers, international enterprises must generate their own unique identifiers. An alternative is to use some unique combination of other attributes as a key. The primary key should be chosen such that its attribute values are never, or very rarely, changed. For instance, the address field of a person should not be part of the primary key, since it is likely to change. Social-security numbers, on the other hand, are guaranteed never to change. Unique identifiers generated by enterprises generally do not change, except if two enterprises merge; in such a case the same identifier may have been issued by both enterprises, and a reallocation of identifiers may be required to make sure they are unique.

It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined.

A relation, say $r1$, may include among its attributes the primary key of another relation, say $r2$. This attribute is called a **foreign key** from $r1$, referencing $r2$.

The relation $r1$ is also called the **referencing relation** of the foreign key dependency, and $r2$ is called the **referenced relation** of the foreign key. For example, the attribute *dept name* in *instructor* is a foreign key from*instructor*, referencing *department*,
since *dept name* is the primary key of *department*. In any database instance, given any tuple, say *ta*, from the *instructor* relation, there must be some tuple, say *tb*, in the *department* relation such that the value of the *dept name* attribute of *ta* is the same as the value of the primary key, *dept name*, of *tb* .

Now consider the *section* and *teaches* relations. It would be reasonable to require that if a section exists for a course, it must be taught by at least one instructor; however, it could possibly be taught by more than one instructor. To enforce this constraint, we would require that if a particular (*course id*, *sec id*, *semester*, *year*) combination appears in *section*, then the same combination must appear in *teaches*. However, this set of values does not form a primary key for *teaches*, since more than one instructor may teach one such section. As a result, we cannot declare a foreign key constraint from *section* to *teaches* (although we can define a foreign key constraint in the other direction, from *teaches* to *section*). The constraint from *section* to *teaches* is an example of a **referential integrity constraint**; a referential integrity constraint requires that the values appearing in specified attributes of any tuple in the referencing relation also appear in specified attributes of at least one tuple in the referenced relation.

## 2.4 Schema Diagrams

A database schema, along with primary key and foreign key dependencies, can be depicted by **schema diagrams**. Figure 2.8 shows the schema diagram for our university organization. Each relation appears as a box, with the relation name at the top in blue, and the attributes listed inside the box. Primary key attributes are shown underlined. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

# Table Joining

*Joining Tables (https://www.zoho.com/reports/help/table/joining-tables.html)*

In a reporting system often you might require to combine data from two or more tables to get the required information for analysis and reporting. To retrieve data from two or more tables, you have to combine the tables through the operation known as "Joining of tables". Joining is a method of establishing a relationship between tables using a common column.

*https://en.wikipedia.org/wiki/Join_(SQL)*

An SQL join clause combines columns from one or more tables in a relational database. It creates a set that can be saved as a table or used as it is. ***A JOIN is a means for combining columns from one (self-join) or more tables by using values common to each***. ANSI-standard SQL specifies five types of JOIN: INNER, LEFT OUTER, RIGHT OUTER, FULL OUTER and CROSS. As a special case, a table (base table, view, or joined table) can JOIN to itself in a self-join.

Relational databases are usually normalized to ***eliminate duplication*** of information such as when entity types have one-to-many relationships. For example, a Department may be associated with a number of Employees. Joining separate tables for Department and Employee effectively creates another table which combines the information from both tables.

All subsequent explanations on join types in this article make use of the following two tables. The rows in these tables serve to illustrate the effect of different types of joins and join-predicates. In the following tables the DepartmentID column of the Department table (which can be designated as Department.DepartmentID) is the primary key, while Employee.DepartmentID is a foreign key.

Employee table

| LastName | DepartmentID |
|---|---|
| Rafferty | 31 |
| Jones | 33 |
| Heisenberg | 33 |
| Robinson | 34 |
| Smith | 34 |
| Williams | NULL |

Department table

| DepartmentID | DepartmentName |
|---|---|
| 31 | Sales |
| 33 | Engineering |
| 34 | Clerical |
| 35 | Marketing |

Note: In the Employee table above, the employee "Williams" has not been assigned to any department yet. Also, note that no employees are assigned to the "Marketing" department.

## *1- Cross join*

CROSS JOIN returns the Cartesian product of rows from tables in the join. In other words, it will produce rows which combine each row from the first table with each row from the second table.[1]

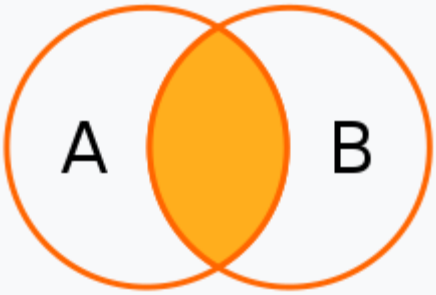
Example of an implicit cross join:

```
SELECT * FROM employee, department;
```

| Employee.LastName | Employee.DepartmentID | Department.DepartmentName | Department.DepartmentID |
|---|---|---|---|
| Rafferty | 31 | Sales | 31 |
| Jones | 33 | Sales | 31 |
| Heisenberg | 33 | Sales | 31 |
| Smith | 34 | Sales | 31 |
| Robinson | 34 | Sales | 31 |
| Williams | NULL | Sales | 31 |
| Rafferty | 31 | Engineering | 33 |
| Jones | 33 | Engineering | 33 |
| Heisenberg | 33 | Engineering | 33 |
| Smith | 34 | Engineering | 33 |
| Robinson | 34 | Engineering | 33 |
| Williams | NULL | Engineering | 33 |
| Rafferty | 31 | Clerical | 34 |
| Jones | 33 | Clerical | 34 |
| Heisenberg | 33 | Clerical | 34 |
| Smith | 34 | Clerical | 34 |
| Robinson | 34 | Clerical | 34 |
| Williams | NULL | Clerical | 34 |
| Rafferty | 31 | Marketing | 35 |
| Jones | 33 | Marketing | 35 |
| Heisenberg | 33 | Marketing | 35 |
| Smith | 34 | Marketing | 35 |
| Robinson | 34 | Marketing | 35 |
| Williams | NULL | Marketing | 35 |

The cross join does not itself apply any predicate to filter rows from the joined table. The results of a cross join can be filtered by using a WHERE clause which may then produce the equivalent of an inner join.

## ***2- Inner join***




A Venn Diagram representing an Inner Join SQL statement between the tables A and B.

An inner join requires each row in the ***two joined tables to have matching column values***, and is a commonly used join operation in applications but should not be assumed to be the best choice in all situations. Inner join creates a new result table by combining column values of two tables (A and B) based upon the join-predicate. The query compares each row of A with each row of B to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied by matching non-NULL values, column values for each matched pair of rows of A and B are combined into a result row.

The "explicit join notation" uses the JOIN keyword, optionally preceded by the INNER keyword, to specify the table to join, and the ON keyword to specify the predicates for the join, as in the following example:

```
SELECT employee.LastName, employee.DepartmentID, department.DepartmentName
FROM employee
INNER JOIN department ON
employee.DepartmentID = department.DepartmentID
```

| Employee.LastName | Employee.DepartmentID | Department.DepartmentName |
|---|---|---|
| Robinson | 34 | Clerical |
| Jones | 33 | Engineering |
| Smith | 34 | Clerical |
| Heisenberg | 33 | Engineering |
| Rafferty | 31 | Sales |

The "implicit join notation" simply lists the tables for joining, in the FROM clause of the SELECT statement, using commas to separate them. Thus it specifies a cross join, and the WHERE clause may apply additional filter-predicates (which function comparably to the join-predicates in the explicit notation).

The following example is equivalent to the previous one, but this time using implicit join notation:

```
SELECT *
FROM employee, department
WHERE employee.DepartmentID = department.DepartmentID;
```

The queries given in the examples above will join the Employee and Department tables using the DepartmentID column of both tables.

Where the DepartmentID of these tables match (i.e. the join-predicate is satisfied), the query will combine the LastName, DepartmentID and DepartmentName columns from the two tables into a result row. Where the DepartmentID does not match, no result row is generated.

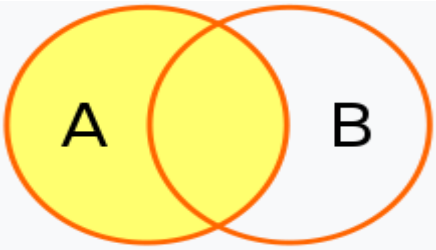
Thus the result of the execution of the query above will be:

| Employee.LastName | Employee.DepartmentID | Department.DepartmentName | Department.DepartmentID |
|---|---|---|---|
| Robinson | 34 | Clerical | 34 |
| Jones | 33 | Engineering | 33 |
| Smith | 34 | Clerical | 34 |
| Heisenberg | 33 | Engineering | 33 |
| Rafferty | 31 | Sales | 31 |

The employee "Williams" and the department "Marketing" do not appear in the query execution results. Neither of these has any matching rows in the other respective table: "Williams" has no associated department, and no employee has the department ID 35 ("Marketing").

Programmers should take special care when joining tables on columns that can contain NULL values, since NULL will never match any other value (not even NULL itself), unless the join condition explicitly uses a combination predicate that first checks that the joins columns are NOT NULL before applying the remaining predicate condition(s).

## *3- Left outer join*




A Venn Diagram representing the Left Join SQL statement between tables A and B.

The result of a *left outer join* (or simply **left join**) for tables A and B always contains all rows of the "left" table (A), even if the join-condition does not find any matching row in the "right" table (B). This means that if the `ON` clause matches 0 (zero) rows in B (for a given row in A), the join will still return a row in the result (for that row) ,but with NULL in each column from B.

A **left outer join** returns all the values from an inner join plus all values in the left table that do not match to the right table, including rows with NULL (empty) values in the link column.

For example, this allows us to find an employee's department, but still shows employees that have not been assigned to a department (contrary to the inner-join example above, where unassigned employees were excluded from the result).

Example of a left outer join (the `OUTER` keyword is optional), with the additional result row (compared with the inner join) italicized:

```
SELECT * FROM employee LEFT OUTER JOIN department ON
employee.DepartmentID = department.DepartmentID;
```

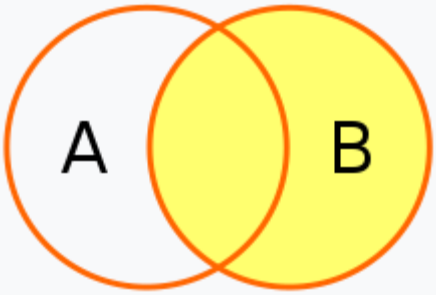
| Employee.LastName | Employee.DepartmentID | Department.DepartmentName | Department.DepartmentID |
|---|---|---|---|
| Jones | 33 | Engineering | 33 |
| Rafferty | 31 | Sales | 31 |
| Robinson | 34 | Clerical | 34 |
| Smith | 34 | Clerical | 34 |
| *Williams* | NULL | NULL | NULL |
| Heisenberg | 33 | Engineering | 33 |

### *Alternative syntaxes*

Oracle supports the deprecated[8] syntax:

```
SELECT *FROM employee, department
WHERE employee.DepartmentID = department.DepartmentID(+)
```

## ***4- Right outer join***




A Venn Diagram representing the Right Join SQL statement between tables A and B.

A **right outer join** (or **right join**) closely resembles a left outer join, except with the treatment of the tables reversed. Every row from the "right" table (B) will appear in the joined table at least once. If no matching row from the "left" table (A) exists, NULL will appear in columns from A for those rows that have no match in B.

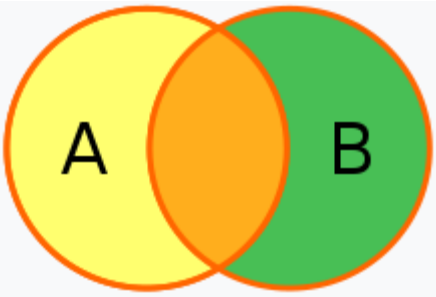
A right outer join returns all the values from the right table and matched values from the left table (NULL in the case of no matching join predicate). For example, this allows us to find each employee and his or her department, but still show departments that have no employees.

Below is an example of a right outer join (the `OUTER` keyword is optional), with the additional result row italicized:

```
SELECT *
FROM employee RIGHT OUTER JOIN department
  ON employee.DepartmentID = department.DepartmentID;
```

| Employee.LastName | Employee.DepartmentID | Department.DepartmentName | Department.DepartmentID |
|---|---|---|---|
| Smith | 34 | Clerical | 34 |
| Jones | 33 | Engineering | 33 |
| Robinson | 34 | Clerical | 34 |
| Heisenberg | 33 | Engineering | 33 |
| Rafferty | 31 | Sales | 31 |
| NULL | NULL | *Marketing* | *35* |

## *5- Full outer join*




A Venn Diagram representing the Full Join SQL statement between tables A and B.

Conceptually, a **full outer join** combines the effect of applying both left and right outer joins. Where rows in the FULL OUTER JOINed tables do not match, the result set will have NULL values for every column of the table that lacks a matching row. For those rows that do match, a single row will be produced in the result set (containing columns populated from both tables).

For example, this allows us to see each employee who is in a department and each department that has an employee, but also see each employee who is not part of a department and each department which doesn't have an employee.

Example of a full outer join (the `OUTER` keyword is optional):

```
SELECT *
FROM employee FULL OUTER JOIN department
  ON employee.DepartmentID = department.DepartmentID;
```

| Employee.LastName | Employee.DepartmentID | Department.DepartmentName | Department.DepartmentID |
|---|---|---|---|
| Smith | 34 | Clerical | 34 |
| Jones | 33 | Engineering | 33 |
| Robinson | 34 | Clerical | 34 |
| *Williams* | NULL | NULL | NULL |
| Heisenberg | 33 | Engineering | 33 |
| Rafferty | 31 | Sales | 31 |
| NULL | NULL | *Marketing* | *35* |

Example: Consider the following three tables :

***teachers table***

```
 id |  name
----+--------
  1 | Volker
  2 | Elke
(2 rows)
```

***Projects table***

```
 id |   name   | duration | teacher
----+----------+----------+---------
  1 | compiler |      180 |       1
  2 | xpaint   |      120 |       1
  3 | game     |      250 |       2
  4 | Perl     |       80 |       4
(4 rows)
```

***Assign table***

```
 project | stud | percentage
---------+------+------------
       1 |    2 |         10
       1 |    4 |         60
       1 |    1 |         30
       2 |    1 |         50
       2 |    4 |         50
       3 |    2 |         70
       3 |    4 |         30
(7 rows)
```

1. SELECT * FROM teachers, projects ***where*** teachers.id = projects.id;

```
 id |  name  | id |   name   | duration | teacher
----+--------+----+----------+----------+---------
  1 | Volker |  1 | compiler |      180 |       1
  2 | Elke   |  2 | xpaint   |      120 |       1
```

2. inner join of tables *teachers* and *project* if the condition is teachers.id ! = projects.id will be

SELECT * FROM teachers, projects where teachers.id != projects.id;

```
 id |  name  | id |   name   | duration | teacher
----+--------+----+----------+----------+---------
  1 | Volker |  2 | xpaint   |      180 |       1
  1 | Volker |  3 | game     |      180 |       1
  1 | Volker |  4 | Perl     |      180 |       1
  2 | Elke   |  1 | compiler |      120 |       1
  2 | Elke   |  3 | game     |      120 |       1
  2 | Elke   |  4 | Perl     |      120 |       1
```

## 6 Indexing and Hashing

Many queries reference only a small proportion of the records in a file. For example , the queries " Find all accounts at the Tech branch" reference only a fraction of the account records. It is inefficient for the system to have to read every record and to check the branch names. The system should be able to locate these records directly. To allow that, we design additional structures with the files.

An index for a file in the system works like a catalog for a book in a library, if we are looking for a book , the catalog of the name of the books tells us where to find the book.

To assist us searching the catalog, the names in the catalog listed in an alphabetic order.

There are two basic kinds of indices :

- Ordered indices: such indices are based on a sorted ordering of the values.
- Hash index : such indices are based on some values , these values calculated by a function called hash function.

We often want to have more than one index for the file. Return to the example of the library, there can be a catalog for the names of the books and another catalog for the others of the books and third one for the subjects of the books.

### *6.1 Ordered Indices :*

These are used to gain fast random access to records in a file. Each index structure is associated with a particular ***search key***. The index stores the values of the search keys in sorted order.

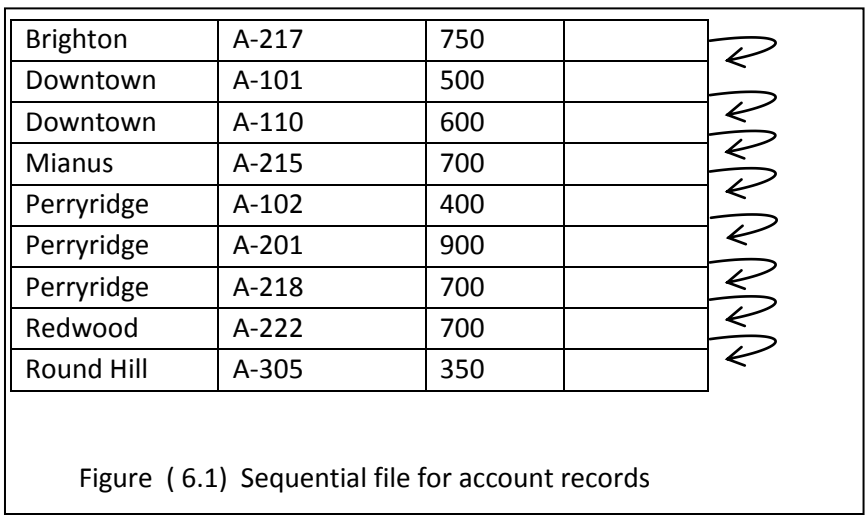
The record in the indexed file may themselves be sorted in some way. The file may have several indices of different search key.

**Primary index :** if the file containing the record is sequentially ordered , the index whose search key specifies the sequential order of the file , this index is a primary index for that file.

**Secondary index :** is the index of the file whose search key specifies an order different from the order of the file.

### 6.1.1 : Ordered Primary Index

Figure 6.1 shows an ordered file for ***account*** records.



| | | | |
|---|---|---|---|
| Brighton | A-217 | 750 | |
| Downtown | A-101 | 500 | |
| Downtown | A-110 | 600 | |
| Mianus | A-215 | 700 | |
| Perryridge | A-102 | 400 | |
| Perryridge | A-201 | 900 | |
| Perryridge | A-218 | 700 | |
| Redwood | A-222 | 700 | |
| Round Hill | A-305 | 350 | |

Figure ( 6.1) Sequential file for account records

The file of figure 6.1 is sorted on a search key order, with branch name is used as the search key.

**6.1.1.1 : Indices Types :** There are two types of ordered indices : ***Dense*** and ***Sparse*** indices.

***Dense index*** : an index entry appears for every search key value in the file. The index record contains the search key value and a pointer to the first data record with that search key value, as shown in figure (6.2) for the account file.

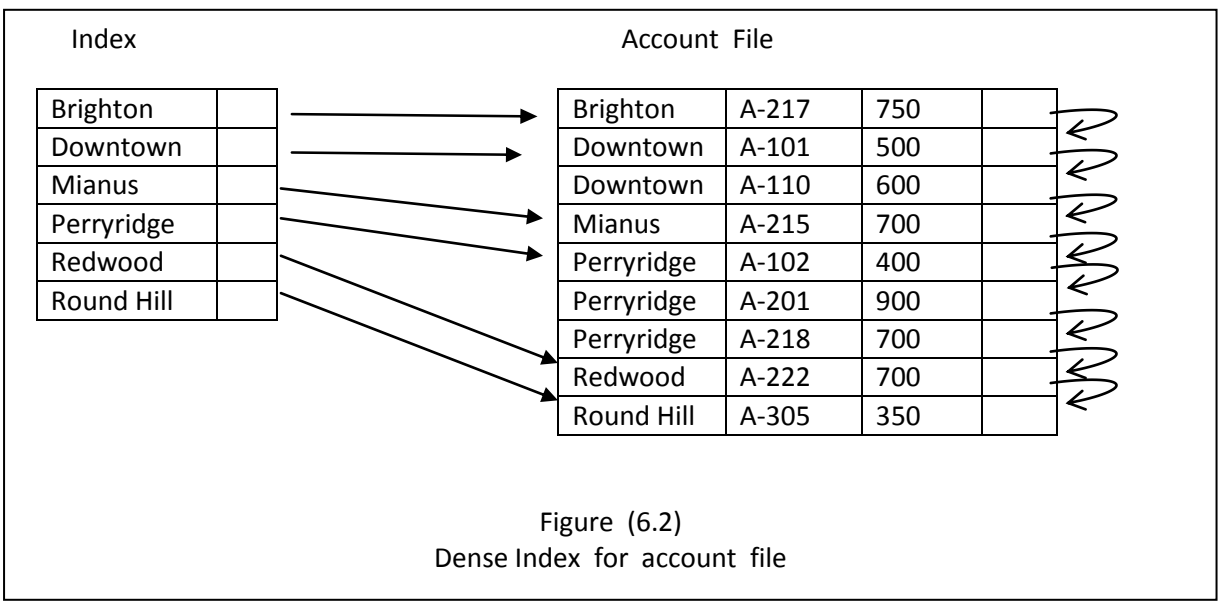



Figure (6.2)
Dense Index for account file

Suppose that we are locking up records for the PEERYIDGE branch using the dense index of figure (6.2), we follow the pointer directly to the first PEERYIDGE record . We process this record and **follow the pointer in that record** to locate the next record in search key (branch name) order. We continue processing records until we encounter a record for a branch other than PEERYIDGE.

***Spars index*** : An index record is created for only some of the values. To locate a record we find the index entry with the largest search key value that is less than or equal to the search key value for which we are locking . We start at the record pointed to by that index entry, and follow the pointer in the file until we find the desired record. As shown in figure (6.3) for the account file.

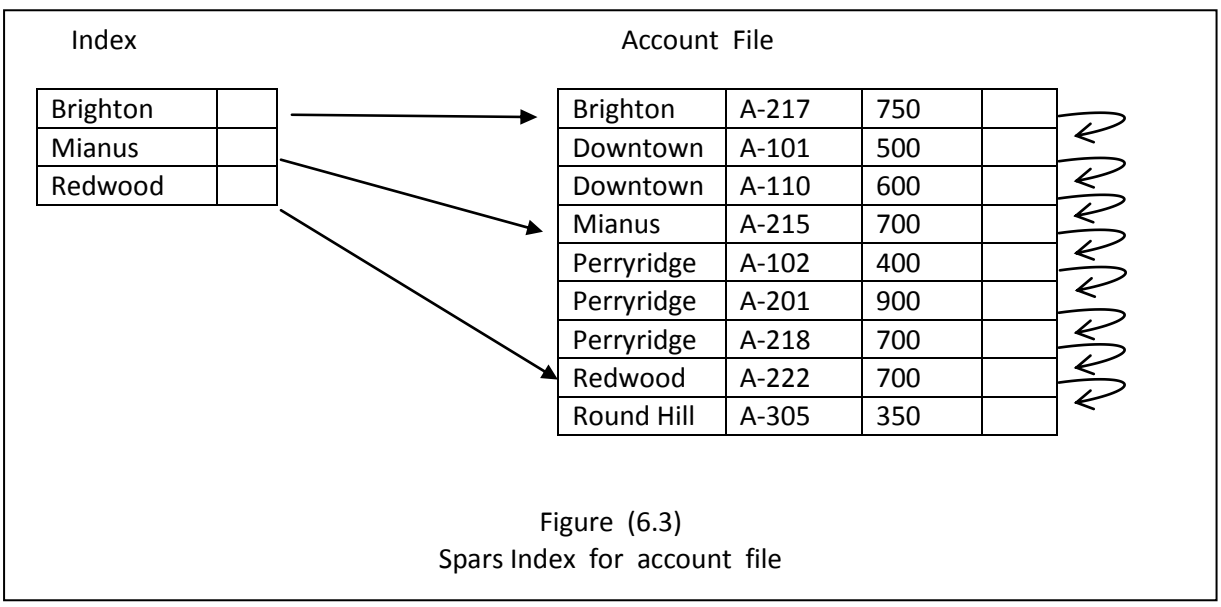



Figure (6.3)
Spars Index for account file

If we are using the spars index of figure (6.3) to find PEERYIDGE , we do not find an index entry for PEERYIDGE in the index. Since the last entry (in alphabetic order) before BEERYIDGE is MIANUS , we follow that pointer.
We then read the account file in sequential order until we find the first PEERYIDEG record and begin processing at that point.

As we have seen , it is faster to locate a record by using a dense index rather than a sparse index., but sparse indices require less space.

#### 6.1.1.2 Index Update

Every index must be updated whenever a record is inserted into the file or deleted from the file .

Deletion : To delete a record we first look up the record to be deleted .

In dense indices ,if the deleted record was the only record with its particular search key value, then the search key value is deleted from the index.

For sparse indices , we delete a key value by replacing its entry (if one exist) in the index with the next search key value . if the next search key value already has an index entry , the entry is deleted instead of being replace.

Insertion : First we perform a lookup using the search key value that appears in the record to be inserted .

In dense index and the value does not appear in the index, the value is inserted in the index.

6.1.2 : Ordered Secondary Index

A secondary index looks like dense primary index except that the records pointed to by successive values in the index are not stored sequentially.
It is not enough to point to just the first record with each search key value because the remaining records with the same search key value could be anywhere in the file, since the records are ordered by a search key of the primary index rather than by the search key of the secondary index. Therefore a secondary index must contain pointers to all the records.
The pointers in the secondary index do not point directly to the file , instead each pointer points to a bucket that contains pointers to the file . Figure (6.4) shows the account file sorted by the balance field, which is not a primary key.

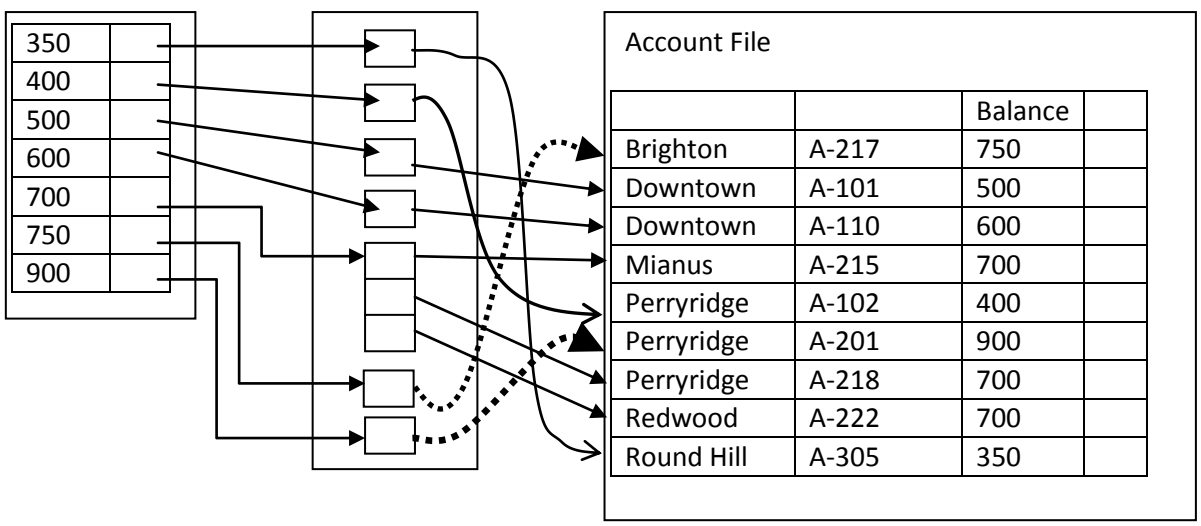



Figure (6.4)
Secondary index for account file on balance as the secondary key

## ***6.2 : Hash Index***

One disadvantage of sequential file is that we must access an index structure to locate data or must use binary search.
File organization based on technique of hashing allow us to avoid accessing an index structure.
Hashing also provide a way of constructing indices.

### *6.2.1 Hash File Organization*

In a hash file organization, we obtain the address of the disk block containing the desired record directly by computing a function on the search key value of the record.

Let ( K ) denote the set of all search key values. Let ( B ) denote the set of all blocks addresses.
A ***hash function ( H )*** is a function from ( K ) to ( B ) .

To insert a record with search key ( Ki ) , we compute [ H(Ki) ] which gives the address of the block for that record and the record stores in that block.

To perform a lockup on the search key value ( Ki ) , we compute [ H(Ki) ] , then search the block with that address. Suppose that to search keys , K5 and K7 , have the same hash value; that is H(K5)=H(K7), thus we have to check the search key value of every record in the block to find the desired record. Figure (6.5) shows Hash organization for the account file.

Block 0

| | | |
|---|---|---|
| | | |

Block 1

| | | |
|---|---|---|
| | | |

Block 2

| | | |
|---|---|---|
| | | |

Block 3

| | | |
|---|---|---|
| Brightton | A-217 | 750 |
| Round Hill | A-305 | 350 |

Block 4

| | | |
|---|---|---|
| Redwood | A-222 | 700 |

Block 5

| | | |
|---|---|---|
| Perryridge | A-102 | 400 |
| Perryridge | A-201 | 900 |
| Perryridge | A-218 | 700 |
| | | |

Block 6

| | | |
|---|---|---|
| | | |

Block 7

| | | |
|---|---|---|
| Mianus | A-215 | 700 |

Block 8

| | | |
|---|---|---|
| Downtown | A-101 | 500 |
| Downtown | A-110 | 600 |

Block 9

| | | |
|---|---|---|
| | | |

Figure (6.5)

### *6.2.2 Hash Index*

Hashing can be used for index structure creation. A hash index organize the search keys with their associated pointers into a hash file structure.

We construct a hash index as follow , we apply a hash function on a search key value to identify the block, and store the key and its pointer in that block as shown in figure (6.6).

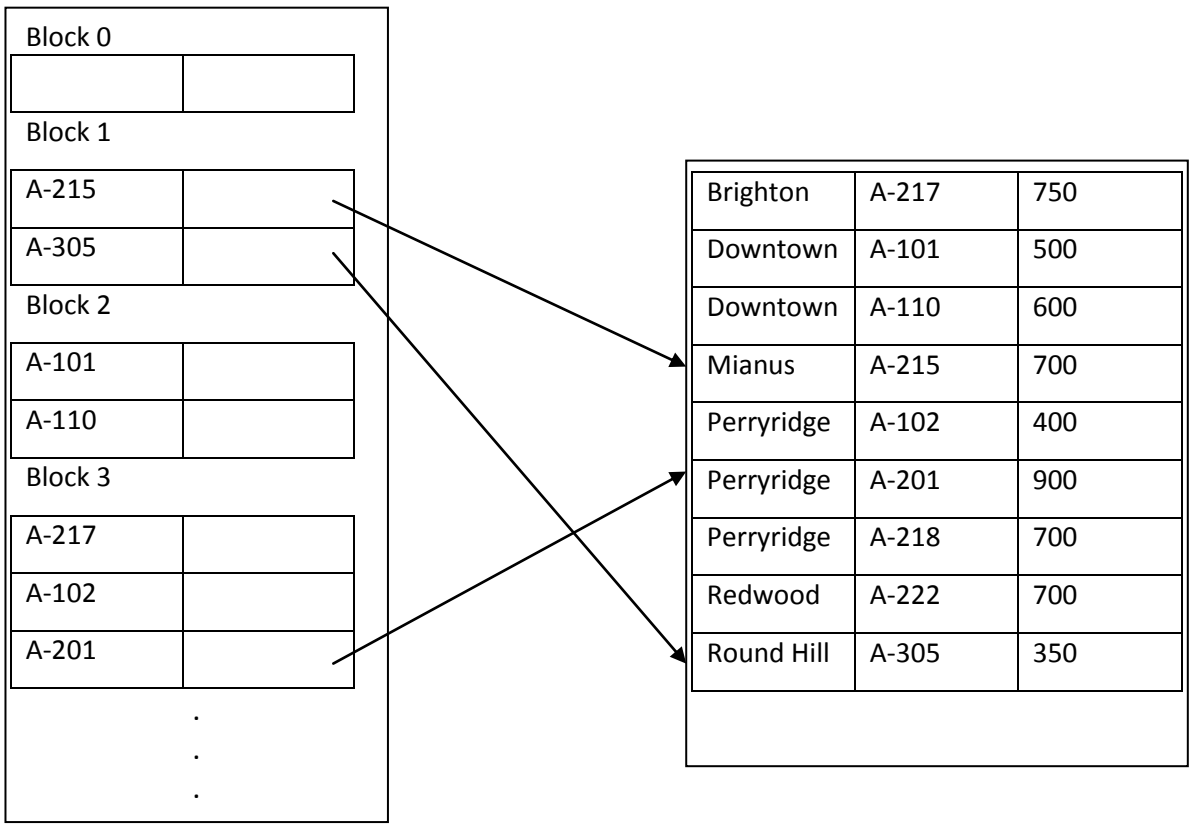



Figure (6.6)
Hash index on search key account-number of account file