Chapter 1: Databases and Database Users

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.

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.

A database can be of any size and of varying complexity. For example, 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.A database may be generated and maintained manually or it may be computerized.

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.

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.

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	04	King
92	CS1310	Fall	04	Anderson
102	CS3320	Spring	05	Knuth
112	MATH2410	Fall	05	Chang
119	CS1310	Fall	05	Anderson
135	CS3380	Fall	05	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Figure 1.2
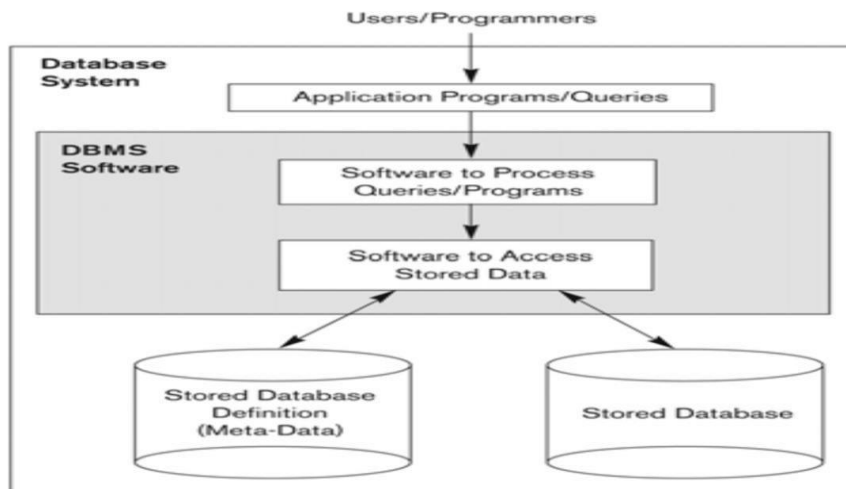A database that stores student and course information.

Figure 1.1
A simplified database
system environment.

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.

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.

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.**

In object-oriented and object-relational databases 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 data model** is a type of data abstraction that is used to provide this conceptual representation

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.

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.

1.4 Actors on the Scene

1.4.1 Database Administrators

1.4.2 Database Designers

1.4.3 End Users

 1.4.4 System Analysts and Application Programmers (Software Engineers)

1.4.1 Database Administrators

In any organization where many persons use the same resources, there is a need for a chief administrator to oversee and manage these resources. In a database environment, the primary resource is the database itself and the secondary resource is the DBMS and related software. Administering these resources is the responsibility of the **database administrator (DBA).** The DBA is responsible for authorizing access to the database, for coordinating and monitoring its use, and for acquiring software and hardware resources as needed.

1.4.2 Database Designers

Database designers are responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data. It is the responsibility of database designers to communicate with all prospective database users, in order to understand their requirements, and to come up with a design that meets these requirements.

1.4.3 End Users

End users are the people whose jobs require access to the database for querying, updating, and generating reports; the database primarily exists for their use. There are several categories of end users:

• **Casual end users** occasionally access the database, but they may need different information each time. They use a sophisticated database query language to specify their requests and are typically middle- or high-level managers or other occasional browsers.

• **Naive** or **parametric end users** make up a sizable portion of database end users. Their main job function revolves around constantly querying and updating the database, using standard types of queries and updates—called **canned transactions**—that have been carefully programmed and tested.

Bank tellers check account balances and post withdrawals and deposits.

• **Sophisticated end users** include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS so as to implement their applications to meet their complex requirements.

- **Stand-alone users** maintain personal databases by using ready-made program packages that provide easy-to-use menu- or graphics-based interfaces. An example is the user of a tax package that stores a variety of personal financial data for tax purposes.

1.4.4 System Analysts and Application Programmers (Software Engineers)

System analysts determine the requirements of end users, especially naive and parametric end users, and develop specifications for canned transactions that meet these requirements. **Application programmers** implement these specifications as programs; then they test, debug, document, and maintain these canned transactions. Such analysts and programmers (nowadays called **software engineers**) should be familiar with the full range of capabilities provided by the DBMS to accomplish their tasks.

1.5 Workers behind the Scene

In addition to those who design, use, and administer a database, others are associated with the design, development, and operation of the DBMS *software and system environment.* These persons are typically not interested in the database itself. We call them the "workers behind the scene," and they include the following categories.

□ □ **DBMS system designers and implementers** are persons who design and implement the DBMS modules and interfaces as a software package. A DBMS is a complex software system that consists of many components or **modules,** including modules for implementing the catalog, query language, interface processors, data access, concurrency control, recovery, and security.

□ □ **Tool developers** include persons who design and implement **tools**—the software packages that facilitate database system design and use, and help improve performance. Tools are optional packages that are often purchased separately. They include packages for database design, performance monitoring, natural language or graphical interfaces, prototyping, simulation, and test data generation .

□ □ **Operators and maintenance personnel** are the system administration personnel who are responsible for the actual running and maintenance of the hardware and software environment for the database system.

1.6 <u>Advantages of Using a DBMS</u>

1.6.1 Controlling Redundancy

 1.6.2 Restricting Unauthorized Access

 1.6.3 Providing Persistent Storage for Program Objects and Data Structures

 1.6.4 Permitting Inferencing and Actions Using Rules

1.6.5 Providing Multiple User Interfaces

1.6.6 Representing Complex Relationships Among Data

1.6.7 Enforcing Integrity Constraints

1.6.8 Providing Backup and Recovery

1.6.9 Additional Implications of the Database Approach

1.6.1 Controlling Redundancy

In traditional software development utilizing file processing, every user group maintains its own files for handling its data-processing applications. For example, consider the UNIVERSITY database example two groups of users might be the course registration personnel and the accounting office. In the traditional approach, each group independently keeps files on students. The accounting office also keeps data on registration and related billing information, whereas the registration office keeps track of student courses and grades. Much of the data is stored twice: once in the files of each user group. Additional user groups may further duplicate some or all of the same data in their own files.

This **redundancy** in storing the same data multiple times leads to several problems. First, there is the need to perform a single logical update—such as entering data on a new student— multiple times: once for each file where student data is recorded. This leads to *duplication of effort*. Second, *storage space is wasted* when the same data is stored repeatedly, and this problem may be serious for large databases. Third, files that represent the same data may become *inconsistent*. This may happen because an update is applied to some of the files but not to others.

In the database approach, the views of different user groups are integrated during database design. For consistency, we should have a database design that stores each logical data item—such as a student's name or birth date—in *only one place* in the database. This does not permit inconsistency, and it saves storage space.

1.6.2 Restricting Unauthorized Access

When multiple users share a database, it is likely that some users will not be authorized to access all information in the database. For example, financial data is often considered confidential, and hence only authorized persons are allowed to access such data. In addition, some users may be permitted only to retrieve data, whereas others are allowed both to retrieve and to update. Hence, the type of access operation—retrieval or update—must also be controlled. Typically, users or user groups are given account numbers protected by passwords, which they can use to gain access to the database.

1.6.3 Providing Persistent Storage for Program Objects and Data Structures

Databases can be used to provide **persistent storage** for program objects and data structures. This is one of the main reasons for the emergence of the **object-oriented database systems.** Programming languages typically have complex data structures, such as record types in PASCAL or class definitions in C++. The values of program variables are discarded once a program terminates.

The persistent storage of program objects and data structures is an important function of database systems. Traditional database systems often suffered from the so-called **impedance mismatch problem,** since the data structures provided by the DBMS were incompatible with the programming language's data structures. Object-oriented database systems typically offer data structure **compatibility** with one or more object-oriented programming languages.

1.6.4 Permitting Inferencing and Actions Using Rules

Some database systems provide capabilities for defining *deduction rules* for *inferencing* new information from the stored database facts. Such systems are called **deductive database systems.** For example, there may be complex rules in the miniworld application for determining

when a student is on probation. These can be specified *declaratively* as **rules,** which when compiled and maintained by the DBMS can determine all students on probation.

1.6.5 Providing Multiple User Interfaces

Because many types of users with varying levels of technical knowledge use a database, a DBMS should provide a variety of user interfaces. These include query languages for casual users; programming language interfaces for application programmers; forms and command codes for parametric users; and menu-driven interfaces and natural language interfaces for stand-alone users. Both forms-style interfaces and menu-driven interfaces are commonly known as **graphical user interfaces (GUIs).**

1.6.6 Representing Complex Relationships Among Data

A database may include numerous varieties of data that are interrelated in many ways. A DBMS must have the capability to represent a variety of complex relationships among the data as well as to retrieve and update related data easily and efficiently.

1.6.7 Enforcing Integrity Constraints

Most database applications have certain **integrity constraints** that must hold for the data. A DBMS should provide capabilities for defining and enforcing these constraints. The simplest type of integrity constraint involves specifying a data type for each data item.

For example we may specify that the value of the Class data item within each student record must be an integer between 1 and 5 and that the value of Name must be a string of no more than 30 alphabetic characters.

1.6.8 Providing Backup and Recovery

A DBMS must provide facilities for recovering from hardware or software failures. The **backup and recovery subsystem** of the DBMS is responsible for recovery. For example, if the computer system fails in the middle of a complex update program, the recovery subsystem is

responsible for making sure that the database is restored to the state it was in before the program started executing.

1.6.9 Additional Implications of the Database Approach

- Potential for Enforcing Standards
- Reduced Application Development Time
- Flexibility
- Availability of Up-to-Date Information
- Economies of Scale

Potential for Enforcing Standards

The Database approach permits the DBA to define and enforce standards among database users in a large organization.Standards can be defined for names and formats of data elements, display formats, report structures, terminology, and so on.

Reduced Application Development Time

A prime selling feature of the database approach is that developing a new application—such as the retrieval of certain data from the database for printing a new report—takes very little time. Designing and implementing a new database from scratch may take more time than writing a single specialized file application

Flexibility

It may be necessary to change the structure of a database as requirements change. Modern DBMSs allow certain types of changes to the structure of the database without affecting the stored data and the existing application programs.

Availability of Up-to-Date Information

A DBMS makes the database available to all users. As soon as one user's update is applied to the database, all other users can immediately see this update. This availability of up-to-date information is essential for many transaction-processing applications, such as reservation systems or banking databases.

Economies of Scale

The DBMS approach permits consolidation of data and applications, thus reducing the amount of wasteful overlap between activities of data-processing personnel in different projects or departments.

1.8 When Not to Use a DBMS

In spite of the advantages of using a DBMS, there are a few situations in which such a system may involve unnecessary overhead costs as that would not be incurred in traditional file processing. The overhead costs of using a DBMS are due to the following:

• High initial investment in hardware, software, and training.

• Generality that a DBMS provides for defining and processing data.

• Overhead for providing security, concurrency control, recovery, and integrity functions.

A brief history of database applications

Early database systems: 1960's to 1970's and 1980's
The main types of early database systems were three types
1.   Hierarchical
2.   Network model
3.   File systems
Large number of records of similar structure were stored and maintained in large organization.

❖ **Drawback**

1.   There was intermixing a conceptual relationship with the physical storage and placement of records on disk. Although for original queries and transaction data access was efficient, it did not provide enough flexibility to access records efficiently when new queries and transactions were identified.

2.   When changes were made to the requirements of the application, it was difficult to reorganize the database.

3.   These systems only provide programming language interfaces.

4.   Implementing new queries and transactions was time-consuming and expensive.

❖   **Relational databases:**

Relational databases could separate the physical storage of the data from its conceptual representation. It could provide a mathematical foundation for databases. It introduces high level query languages that provided an alternative to programming language interfaces.
It was developed in the late 1970's and the commercial RDBMS was introduced in the early 1980's.

→**Advantages**

1.   Introduction of high level query language made it easier to write new queries and recognize database as required.

2.   They did not use physical storage pointers and record placement to access related data records.

3.   Performance greatly improved with the development of new storage and indexing.

4.   Query processing became better.

5.   They are widely used for traditional database systems.

❖   **Object-Oriented databases**

In the 1980's with the emergence of object oriented programming languages, it was necessary to store and share complex structured objects. This led to the development

of object oriented databases. They are used in specialized applications such as engineering design, multimedia publishing and manufacturing systems.

➔ **Advantages**

1.   It provided more general data structures.

2.   In incorporated many of the useful object oriented paradigms, such as ADT(Abstract Data Types), encapsulation, inheritance etc..

❖ **Web based database applications**

The World Wide Web is a large interconnection of a number of computer networks. User can create web documents (using HTML (Hyper Text Markup Language)) called web pages and store them on web servers from where other web clients can access. Documents can be linked together through hyperlinks, which are pointers to other documents.

❖ *Advanced database applications*

Some of the advanced applications of database systems are

1.   **Multimedia databases:** that can store pictures, video clips and sound messages.

2.   **Geographic information systems (GIS):** that can store and analyze maps, weather data and satellite images.

3.   **Data warehouse and online analytical processing (OLAP):**that are used in many companies extract and analyze useful information from very large databases for decision making.

4.   **Real time and active database technology:** that is used in controlling industrial and manufacturing processes.

5.   **Mobile databases:** they are available on the user portable computers, users interact with the mobile database application, which in turn accesses the data stored in the mobile databases through the DBMS.

6.   **Web databases:** databases are integrated with web to support business operation like e-commerce supply chain management or web publishing.

7.   **Spatial database:** provide concepts for database that keep track of objects in a multidimensional space. Ex: Data in CAD/CAM computers.

8.   **Time series:** these applications store information such as economic data at regular points in time like daily sales or monthly sales.

Basic relational systems were not very suitable for many of these applications for the reasons:

1.   More complex data structures were needed for modelling the application.

2.   New data types were needed in addition to standard data types.

3.   New operations and query language constructs were necessary manipulate with the new data types.

4.   New storage and indexing structures were needed.

Chapter 2: Database System Concepts and Architecture

A **data model**—a collection of concepts that can be used to describe the structure of a database—provides the necessary means to achieve this abstraction .By *structure of a database* we mean the data types, relationships, and constraints that should hold on the data. Most data models also include a set of **basic operations** for specifying retrievals and updates on the database.

This allows the database designer to specify a set of valid **user-defined operations** that are allowed on the database objects .An example of a user-defined operation could be COMPUTE_GPA, which can be applied to a STUDENT object.

2.1.1 Categories of Data Models

A) **High-level** or **conceptual data models** provide concepts that are close to the way many users perceive data.

B) **low-level** or **physical data models** provide concepts that describe the details of how data is stored in the computer.

C) representational (or **implementation**) **data models,** which provide concepts that may be understood by end users but that are not too far removed from the way data is organized within the computer. Representational data models represent data by using record structures and hence are sometimes called **record-based data models.**

An **entity** represents a real-world object or concept, such as an employee or a project, that is described in the database. An **attribute** represents some property of interest that further describes an entity, such as the employee's name or salary. A **relationship** among two or more entities represents an interaction among the entities; for example, a works-on relationship between an employee and a project.

D) Representational or implementation data models are the models used most frequently in traditional commercial DBMSs, and they include the widely-used **relational data model.**

E) legacy data models—the **network** and **hierarchical models.**

F) **object data models** are new family of higher-level implementation data models that are closer to conceptual data models.

2.1.2 Schemas, Instances, and Database State

In any data model it is important to distinguish between the *description* of the database and the *database itself.* The description of a database is called the **database schema,** which is specified during database design and is not expected to change frequently.

A displayed schema is called a **schema diagram.** A schema diagram displays only *some aspects* of a schema, such as the names of record types and data items, and some types of constraints.

STUDENT

Name	Student_number	Class	Major

Figure 2.1
Schema diagram for the
database in Figure 1.2.

COURSE

Course_name	Course_number	Credit_hours	Department

PREREQUISITE

Course_number	Prerequisite_number

SECTION

Section_identifier	Course_number	Semester	Year	Instructor

GRADE_REPORT

Student_number	Section_identifier	Grade

The data in the database at a particular moment in time is called a **database state** or **snapshot.** It is also called the *current* set of **occurrences** or **instances** in the database.

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	04	King
92	CS1310	Fall	04	Anderson
102	CS3320	Spring	05	Knuth
112	MATH2410	Fall	05	Chang
119	CS1310	Fall	05	Anderson
135	CS3380	Fall	05	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Figure 1.2
A database that stores student and course information.

The DBMS is partly responsible for ensuring that *every* state of the database is a **valid state**— that is, a state that satisfies the structure and constraints specified in the schema. Hence, specifying a correct schema to the DBMS is extremely important, and the schema must be designed with the utmost care. The DBMS stores the descriptions of the schema constructs and constraints—also called the **meta-data**—in the DBMS catalog so that DBMS software can refer to the schema whenever it needs to. The schema is sometimes called the **intension,** and a database state an **extension** of the schema.

2.2 DBMS Architecture and Data Independence

In this section we specify an architecture for database systems, called the **three-schema architecture** which was proposed to help achieve and visualize the DBMS characteristics.

2.2.1 The Three-Schema Architecture

Proposed to support DBMS characteristics of:

.**Program-data independence.**

.Support of **multiple views** of the data.

.Not explicitly used in commercial DBMS products,but has been useful in explaining databasesystem organization

Figure 2.2
The three-schema architecture.

External Level

External/Conceptual Mapping

Conceptual Level

Conceptual/Internal Mapping

Internal Level

End Users

External View

External View

Conceptual Schema

Internal Schema

Stored Database
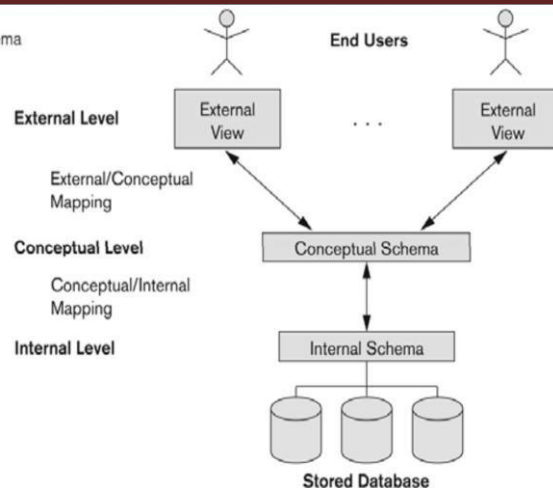
The goal of the three-schema architecture, illustrated in Figure.

1. The **internal level** has an **internal schema,** which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.

2. The **conceptual level** has a **conceptual schema,** which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.

3. The **external** or **view level** includes a number of **external schemas** or **user views.** Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The processes of transforming requests and results between levels are called **mappings.** These mappings may be time-consuming, so some DBMSs—especially those that are meant to support small databases—do not support external

views. Even in such systems, however, a certain amount of mapping is necessary to transform requests between the conceptual and internal levels.

2.2.2 Data Independence

The three-schema architecture can be used to explain the concept of **data independence,** which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), or to reduce the database (by removing a record type or data item).

2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual (or external) schemas. Changes to the internal schema may be needed because some physical files had to be reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema .

2.3 Database Languages and Interfaces

2.3.1 DBMS Languages

Data definition language (**DDL**)is used by the DBA and by database designers to define both schemas. The DBMS will have a DDL compiler whose function is to process DDL statements in order to identify descriptions of the schema constructs and to store the schema description in the DBMS catalog.the DDL is used to specify the conceptual schema only.

Storage definition language (SDL)is used to specify the internal schema. The mappings between the two schemas may be specified in either one of these languages.

View definition language (VDL) is used to specify user views and their mappings to the conceptual schema, but in most DBMSs the DDL is used to define both conceptual and external schemas.

Data manipulation language (DML) Once the database schemas are compiled and the database is populated with data, users must have some means to manipulate the database. Typical manipulations include retrieval, insertion, deletion, and modification of the data. The DBMS provides a **data manipulation language (DML)** for these purposes.There are two main types of DMLs. A **high-level** or **nonprocedural DML** can be used on its own to specify complex database operations in a concise manner.A **low-level** or **procedural DML** *must* be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects from the database and processes each separately.Low-level DMLs are also called **record-at-a-time DMLs.**High-level DMLs, such as SQL, can specify and retrieve many records in a single DML statement and are hence called **set-at-a-time** or **set-oriented DMLs.**

2.3.2 DBMS Interfaces

➢   Menu-Based Interfaces for Browsing

➢   Forms-Based Interfaces

➢   Graphical User Interfaces

➢   Natural Language Interfaces Interfaces for Parametric Users

➢   Interfaces for the DBA

Menu-Based Interfaces for Browsing

These interfaces present the user with lists of options, called **menus,** that lead the user through the formulation of a request. Menus do away with the need to memorize the specific commands and syntax of a query language; rather, the query is composed step by step by picking options from a menu that is displayed by the system.They are often used in **browsing interfaces,** which allow a user to look through the contents of a database in an exploratory and unstructured manner.

Forms-Based Interfaces

A forms-based interface displays a **form** to each user. Users can fill out all of the form entries to insert new data, or they fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Forms are usually designed and programmed for naive users as interfaces to canned transactions.

Graphical User Interfaces

A graphical interface (GUI) typically displays a schema to the user in diagrammatic form. The user can then specify a query by manipulating the diagram. In many cases, GUIs utilize both menus and forms. Most GUIs use a **pointing device,** such as a mouse, to pick certain parts of the displayed schema diagram.

Natural Language Interfaces

These interfaces accept requests written in English or some other language and attempt to "understand" them. A natural language interface usually has its own "schema," which is similar to the database conceptual schema. The natural language interface refers to the words in its schema, as well as to a set of standard words, to interpret the request.

Interfaces for Parametric Users

Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. Systems analysts and programmers design and implement a special interface for a known class of naive users. Usually, a small set of abbreviated commands is included, with the goal of minimizing the number of keystrokes required for each request

Interfaces for the DBA

Most database systems contain privileged commands that can be used only by the DBA's staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structures of a database.

2.4 The Database System Environment

2.4.1 DBMS Component Modules
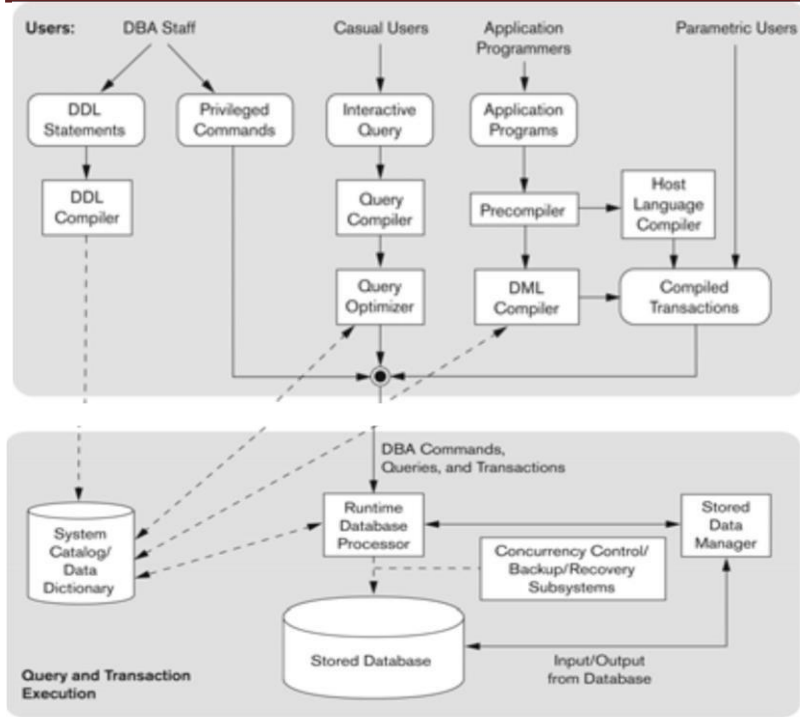
Figure 2.38
Component modules of a DBMS and their interactions.

The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the **operating system (OS),** which schedules disk input/output. A higher-level **stored data manager** module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog.The stored data manager may use basic OS services for carrying out low-level data transfer between the disk and computer main storage, but it controls other aspects of data transfer, such as handling buffers in main memory.

The **DDL compiler** processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog. The catalog includes information such as the names of files, data items, storage details of each file and so on.

The **run-time database processor** handles database accesses at run time; it receives retrieval or update **query compiler** handles high-level queries that are entered interactively. It parses, analyzes, and compiles or interprets a query by creating database access code, and then generates calls to the run-time processor for executing the code.

The **pre-compiler** extracts DML commands from an application program written in a host programming language. These commands are sent to the **DML compiler** for compilation into

object code for database access. The rest of the program is sent to the host language compiler. The object codes for the DML commands and the rest of the program are linked, forming a canned transaction whose executable code includes calls to the runtime database processor.

2.4.2 Database System Utilities

Most DBMSs have **database utilities** that help the DBA in managing the database system.

Common utilities have the following types of functions:

1. Loading: A loading utility is used to load existing data files—such as text files or sequential files—into the database. Usually, the current (source) format of the data file and the desired (target) database file structure are specified to the utility.

2. Backup: A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape. The backup copy can be used to restore the database in case of catastrophic failure. Incremental backups are also often used, where only changes since the previous backup are recorded. Incremental backup is more complex but it saves space.

3. File reorganization: This utility can be used to reorganize a database file into a different file organization to improve performance.

4. Performance monitoring: Such a utility monitors database usage and provides statistics to the DBA. The DBA uses the statistics in making decisions such as whether or not to reorganize files to improve performance .

2.4.3 Tools, Application Environments, and Communications Facilities

Other tools are often available to database designers, users, and DBAs. **CASE tools a**re used in the design phase of database systems. Another tool that can be quite useful in large organizations is an expanded **data dictionary** (or **data repository**) **system.**In addition to storing catalog information about schemas and constraints, the data dictionary stores other information, such as design decisions, usage standards, application program descriptions, and user information. Such a system is also called an **information repository.**

Application development environments, such as the PowerBuilder system, are becoming quite popular.

The DBMS also needs to interface with **communications software,** whose function is to allow users at locations remote from the database system site to access the database through computer terminals, workstations, or their local personal computers.These are connected to the database site through data communications hardware such as phone lines, long-haul networks, local-area networks, or satellite communication devices.

2.5 . DBMS architecture

a)Centralized architecture

.Combines everything into single system including-DBMS software, hardware, application programs,and user interface processing software.

.User can still connect through a remote terminal –however, all processing is done at centralized site.
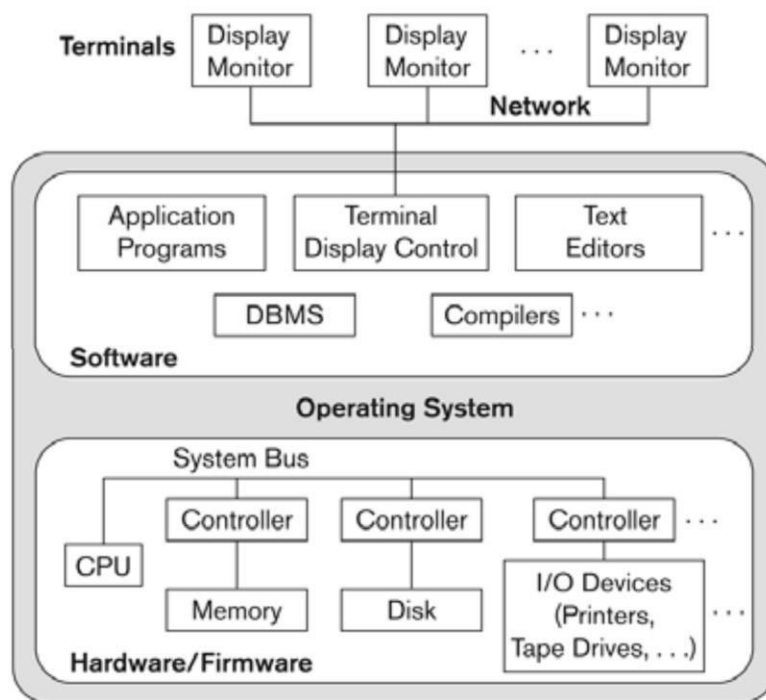
Figure 2.4. A physical centralized architecture..

b) Basic 2-tier Client-Server Architectures

Specialized Servers with Specialized functions

        Print server

        File server

        DBMS server

        Web server

        Email server

        Clients can access the specialized servers asNeeded

Logical two-tier client server architecture

Figure 2.5
Logical two-tier client/server architecture.
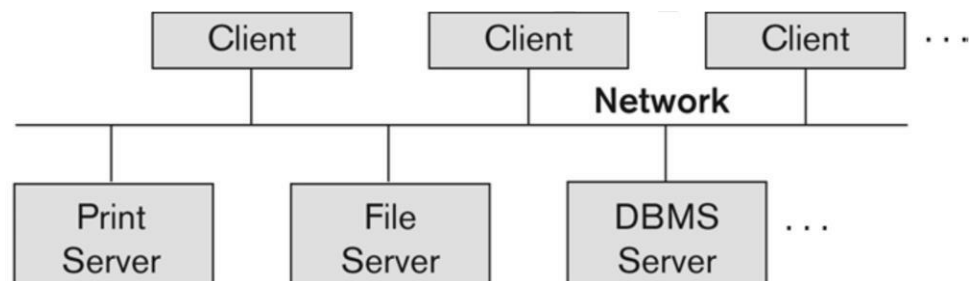
Clients

Provide appropriate interfaces through a clientsoftware module to access and utilize the various servers resources.

Clients may be diskless machines or PCs orWorkstations with disks with only the clientsoftware installed.

Connected to the servers via some form of anetwork.

(LAN: local area network, wireless network, etc.)

DBMS SERVER

. Provides database query and transaction services to theclients..

.Relational DBMS servers are often called SQL servers,query servers, or transaction servers
Applications running on clients utilize an Application program Interface (**API**) to access server databasesviastandard interface such as:

□ ODBC: Open Database Connectivity standard

☐ JDBC: for Java programming access

   . Client and server must install appropriate client moduleand server module software for ODBC or JDBC

A client program may connect to several DBMSs,sometimes called the data sources.

☐In general, data sources can be files or othernon-DBMS software that manages data.

☐Other variations of clients are possible: e.g., insome object DBMSs, more functionality istransferred

Toclients including data dictionaryfunctions, optimization and recovery acrossmultiple servers, etc.'

c) Three-tier client /server Architecture

Common for Web applications

1)Intermediate Layer called Application Server or WebServer: Stores the web connectivity software and the business logicpart of the application used to access the correspondingdata from the database server..Acts like a conduit for sending partially processed databetween the database server and the client.

2).Three-tier Architecture Can Enhance Security:

.Database server only accessible via middle tier

. Clients cannot directly access database server

Figure 2.7
Logical three-tier
client/server architecture,
with a couple of commonly
used nomenclatures.

Client	GUI, Web Interface	Presentation Layer
Application Server or Web Server	Application Programs, Web Pages	Business Logic Layer
Database Server	Database Management System	Database Services Layer
	(a)	(b)
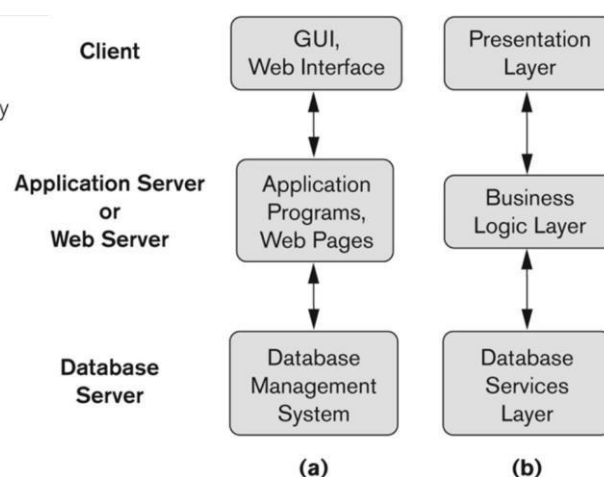
2.6 **Classification of Database Management Systems**

Several criteria are normally used to classify DBMSs.

The first is the **data model** on which the DBMS is based. The two types of data models used in many current commercial DBMSs are the **relational data model** and the **object data model.** Many legacy applications still run on database systems based on the **hierarchical** and **network data models.**The relational DBMSs are evolving continuously, and, in particular, have been incorporating many of the concepts that were developed in object databases. This has led to a new class of DBMSs that are being called **object-relational DBMSs/.**We can hence categorize DBMSs based on the data model: relational, object, object-relational, hierarchical, network, and other.

The second criterion used to classify DBMSs is the **number of users** supported by the system. **Single-user systems** support only one user at a time and are mostly used with personal computers. **Multiuser systems,** which include the majority of DBMSs, support multiple users concurrently.

A third criterion is the **number of sites** over which the database is distributed. A DBMS is **centralized** if the data is stored at a single computer site. A centralized DBMS can support multiple users, but the DBMS and the database themselves reside totally at a single computer site. A **distributed DBMS (DDBMS)** can have the actual database and DBMS software distributed over many sites, connected by a computer network. **Homogeneous DDBMSs** use the same DBMS software at multiple sites.

A fourth criterion is the **cost** of the DBMS. The majority of DBMS packages cost between $10,000 and $100,000. Single-user low-end systems that work with microcomputers cost between $100 and $3000. At the other end, a few elaborate packages cost more than $100,000.

A fifth criteria is on the basis of the **types of access path** options for storing files. One well-known family of DBMSs is based on inverted file structures.

Finally DBMS can be **general-purpose** or **special-purpose.** When performance is a primary consideration, a special-purpose DBMS can be designed and built for a specific application; such a system cannot be used for other applications without major changes.

Ex:Many airline reservations and telephone directory systems developed in the past are special-purpose DBMSs. These fall into the category of **on-line transaction processing (OLTP) systems,** which must support a large number of concurrent transactions without imposing excessive delays.

## Chapter 3: Data Modeling Using the Entity-Relationship Model

Conceptual modeling is an important phase in designing a successful database application. Generally, the term **database application** refers to a particular database—for example, a BANK database that keeps track of customer accounts—and the associated *programs* that implement the database queries and updates—for example, programs that implement database updates corresponding to customers making deposits and withdrawals. These programs often provide user-friendly graphical user interfaces (GUIs) utilizing forms and menus. Hence, part of the database application will require the design, implementation, and testing of these **application programs.**

Database design methodologies attempt to include more of the concepts for specifying operations on database objects, and as software engineering methodologies specify in more detail the structure of the databases that software programs will use and access, it is certain that this commonality will increase .

Entity-Relationship (ER) model, which is a popular high-level conceptual data model. This model and its variations are frequently used for the conceptual design of database applications, and many database design tools employ its concepts. We describe the basic data-structuring concepts and constraints of the ER model and discuss their use in the design of conceptual schemas for database applications.

1. **Using High-Level Conceptual Data Models for Database Design**

Figure shows a simplified description of the database design process. The first step shown is **requirements collection and analysis.** During this step, the database designers interview prospective database users to understand and document their **data requirements.** The result of this step is a concisely written set of users' requirements. These requirements should be specified in as detailed and complete a form as possible. In parallel with specifying the data requirements, it is useful to specify the known **functional requirements** of the application. These consist of the user-defined **operations** (or **transactions**) that will be applied to the database, and they include both retrievals and updates.

Figure 3.1
A simplified diagram
to illustrate the
main phases of
database design.
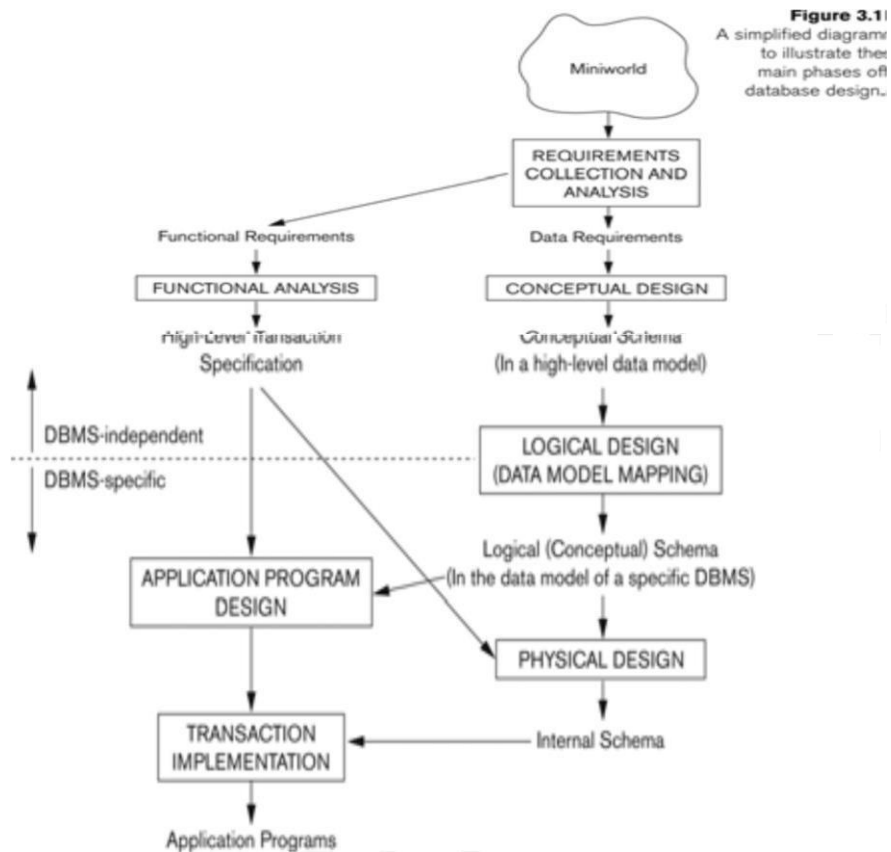
Once all the requirements have been collected and analyzed, the next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This step is called **conceptual design.** The conceptual schema is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model. Because these concepts do not include implementation details, they are usually easier to understand and can be used to communicate with nontechnical users..

During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user operations identified during functional analysis. This also serves to

confirm that the conceptual schema meets all the identified functional requirements. Modifications to the conceptual schema can be introduced if some functional requirements cannot be specified in the initial schema.

The next step in database design is the actual implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model—such as the relational or the object database model—so the conceptual schema is transformed from the high-level data model into the implementation data model. This step is called **logical design** or **data model mapping,** and its result is a database schema in the implementation data model of the DBMS.

Finally, the last step is the **physical design** phase, during which the internal storage structures, access paths, and file organizations for the database files are specified. In parallel with these activities, application programs are designed and implemented as database transactions corresponding to the high-level transaction specifications

2 <u>An Example Database Application</u>

In this section we describe an example database application, called COMPANY, which serves to illustrate the ER model concepts and their use in schema design.

1. The company is organized into departments. Each department has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.

2. A department controls a number of projects, each of which has a unique name, a unique number, and a single location.

3. We store each employee's name, social security number (Note 1), address, salary, sex, and birth date. An employee is assigned to one department but may work on several projects, which are not necessarily controlled by the same department. We keep track of the number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee.

4. We want to keep track of the dependents of each employee for insurance purposes. We keep each dependent's first name, sex, birth date, and relationship to the employee.

3 Entity Types, Entity Sets, Attributes, and Keys

The ER model describes data as entities, relationships, and attributes.

Entities

The basic object that the ER model represents is an **entity,** which is a "thing" in the real world with an independent existence. An entity may be an object with a physical existence—a particular person, car, house, or employee—or it may be an object with a conceptual existence—a company, a job, or a university course.

Attributes : Each entity has **attributes**—the particular properties that describe it. For example, an employee entity may be described by the employee's name, age, address, salary, and job.

A particular entity will have a **value** for each of its attributes. The attribute values that describe each entity become a major part of the data stored in the database.

Ex: 1)The employee entity e1 has four attributes: Name, Address, Age, and HomePhone; their values are "John Smith," "2311 Kirby, Houston, Texas 77001," "55," and "713-749-2630," respectively.

Several types of attributes occur in the ER model:

1)   *simple* versus *composite*

2)   *single-valued* versus *multi valued*

3)   *stored*versus *derived*.

1) **Composite attributes** can be divided into smaller subparts, which represent more basic attributes with independent meanings.

Ex: Address attribute of the employee entity can be sub-divided into StreetAddress, City, State, and Zip with the values "2311 Kirby," "Houston," "Texas," and "77001."

Figure 3.4
A hierarchy of composite attributes.
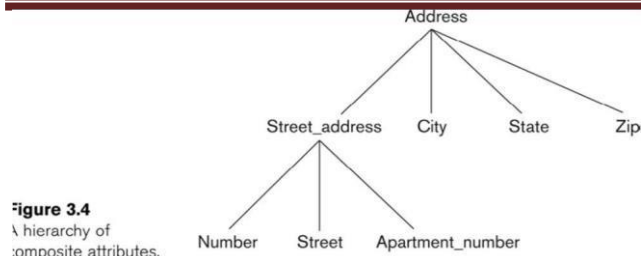
Composite attributes can form a hierarchy; for example, StreetAddress can be subdivided into three simple attributes, Number, Street, and ApartmentNumber, as shown in Figure .The value of a composite attribute is the concatenation of the values of its constituent simple attributes.

2)simpleor **atomic attributes :**

Attributes that are not divisible are called **simple** or **atomic attributes.**

          EX: Number, street are simple attributes

3)Single-valued

Most attributes have a single value for a particular entity; such attributes are called **single-valued.**

          EX: Age is a single-valued attribute of person.

4) Multivalued Attributes

In some cases an attribute can have a set of values for the same entity—for example, a Colors attribute for a car, or a CollegeDegrees attribute for a person. Cars with one color have a single value, whereas two-tone cars have two values for Colors. A multivalued attribute may have lower and upper bounds on the number of values allowed for each individual entity.

Ex:The Colors attribute of a car may have between one and three values, if we assume that a car can have at most three colors.

5)Stored Versus Derived Attributes

In some cases two (or more) attribute values are related—for example, the Age and BirthDate attributes of a person. For a particular person entity, the value of Age can be determined from the current (today's) date and the value of that person's BirthDate. The Age attribute is hence called a **derived attribute** and is said to be **derivable from** the BirthDate attribute, which is called a **stored attribute.**

6) Null Values

In some cases a particular entity may not have an applicable value for an attribute. For example, the ApartmentNumber attribute of an address applies only to addresses that are in apartment buildings and not to other types of residences, such as single-family homes. For such situations, a special value called **null** is created. An address of a single-family home would have null for its ApartmentNumber attribute. Null can also be used if we do not know the value of an attribute for a particular entity—for example, if we do not know the home phone of "John Smith" .The meaning of the former type of null is *not applicable*, whereas the meaning of the latter is *unknown*.

The unknown category of null can be further classified into two cases. The first case arises when it is known that the attribute value exists but is *missing*—for example, if the Height attribute of a person is listed as null.

The second case arises when it is *not known* whether the attribute value exists—for example, if the HomePhone attribute of a person is null.

7) Complex Attributes

Notice that composite and multivalued attributes can be nested in an arbitrary way. We can represent arbitrary nesting by grouping components of a composite attribute between parentheses ( ) and separating the components with commas, and by displaying multivalued attributes between braces {}. Such attributes are called **complex attributes.**

1) For example, PreviousDegrees of a STUDENT is acomposite multi-valued attribute denoted by

        {PreviousDegrees (College, Year, Degree, Field)}

2) If a person can have more than one residence and each residence can have multiple phones, an attribute AddressPhone for a PERSON entity type can be specified as complex attribute.

Entity Types

A database usually contains groups of entities that are similar. For example, a company employing hundreds of employees may want to store similar information concerning each of the

employees. These employee entities share the same attributes, but each entity has *its own value(s)* for each attribute.

An **entity type** defines a *collection* (or *set*) of entities that have the same attributes. Each entity type in the database is described by its name and attributes.

Ex:Two entity types, named EMPLOYEE and COMPANY, and a list of attributes for each.

Entity Sets

The collection of all entities of a particular entity type in the database at any point in time is called an **entity set;** the entity set is usually referred to using the same name as the entity type. For example, EMPLOYEE refers to both a *type of entity* as well as the current *set of all employee entities* in the database.

An entity type is represented in ER diagrams as a rectangular box enclosing the entity type name. Attribute names are enclosed in ovals and are attached to their entity type by straight lines. Composite attributes are attached to their component attributes by straight lines. Multivalued attributes are displayed in double ovals.

An entity type describes the **schema** or **intension** for a *set of entities* that share the same structure. The collection of entities of a particular entity type are grouped into an entity set, which is also called the **extension** of the entity type.

Key Attributes of an Entity Type

An important constraint on the entities of an entity type is the **key** or **uniqueness constraint** on attributes. An entity type usually has an attribute whose values are distinct for each individual entity in the collection. Such an attribute is called a **key attribute,** and its values can be used to identify each entity uniquely.

Ex: 1)The Name attribute is a key of the COMPANY entity type , because no two companies are allowed to have the same name.

2)For the PERSON entity type, a typical key attribute is SocialSecurityNumber.

Sometimes, several attributes together form a key, meaning that the *combination* of the attribute values must be distinct for each entity. If a set of attributes possesses this property, we can define a *composite attribute* that becomes a key attribute of the entity type.

In ER diagrammatic notation, each key attribute has its name **underlined** inside the oval.

Specifying that an attribute is a key of an entity type means that the preceding uniqueness property must hold for *every extension* of the entity type. Hence, it is a constraint that prohibits any two entities from having the same value for the key attribute at the same time.

Some entity types have *more than one* key attribute.

Figure 3.7
The CAR entity type with two key attributes, Registration and Vehicle_id. (a) ER diagram notation. (b) Entity set with three entities.
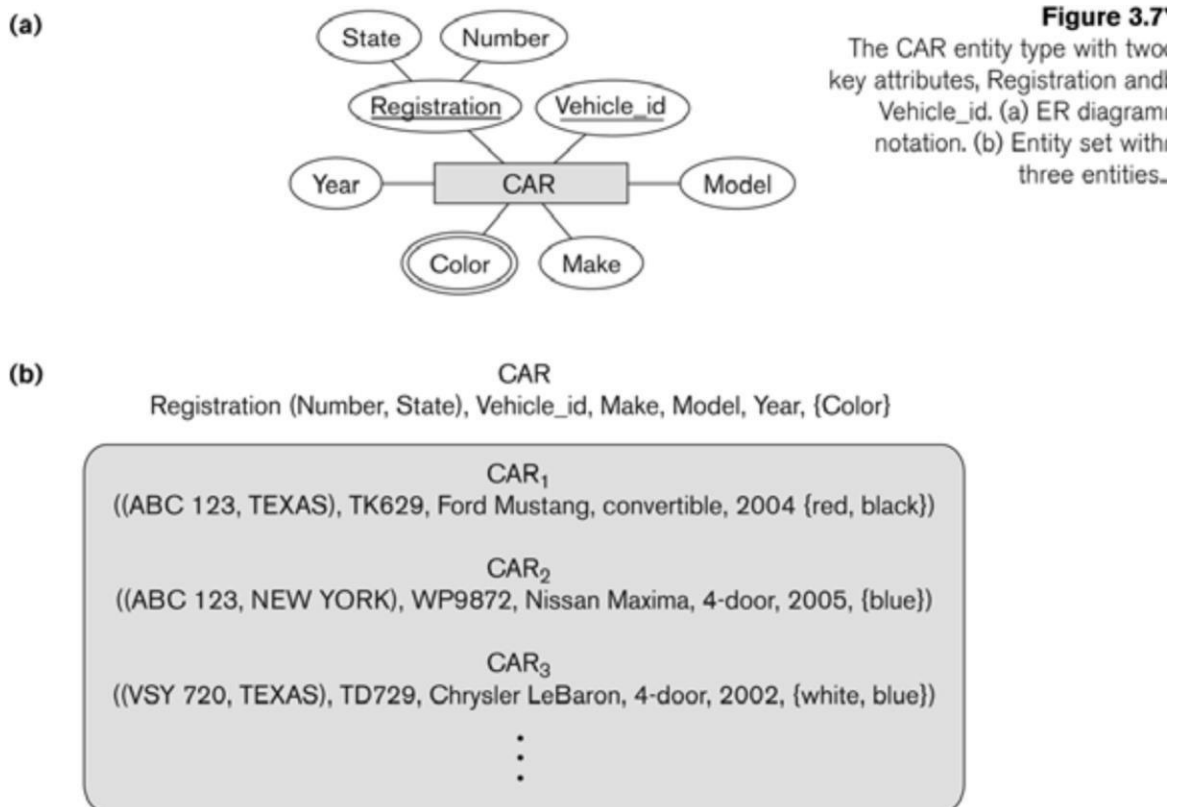
For example, each of the VehicleID and Registration attributes of the entity type CAR is a key in its own right. The Registration attribute is an example of a composite key formed from two simple component attributes, RegistrationNumber and State, neither of which is a key on its own. An entity type may also have *no key,* in which case it is called a *weak entity type* .

Value Sets (Domains) of Attributes

Each simple attribute of an entity type is associated with a **value set** (or **domain** of values), which specifies the set of values that may be assigned to that attribute for each individual entity.

Ex:The range of ages allowed for employees is between 16 and 70, we can specify the value set of the Age attribute of EMPLOYEE to be the set of integer numbers between 16 and 70.

Similarly, we can specify the value set for the Name attribute as being the set of strings of alphabetic characters separated by blank characters and so on. Value sets are not displayed in ER diagrams.

4 Relationships, Relationship Types, Roles, and Structural Constraints

There are several *implicit relationships* among the various entity types. In fact, whenever an attribute of one entity type refers to another entity type, some relationship exists.

For example, the attribute Manager of DEPARTMENT refers to an employee who manages the department.In the ER model, these references should not be represented as attributes but as **relationships.**

4.1 Relationship Types, Sets and Instances

A **relationship type** R among n entity types , , . . ., defines a set of associations—or a **relationship set**—among entities from these types. As for entity types and entity sets, a relationship type and its corresponding relationship set are customarily referred to by the *same name R.* Mathematically, the relationship set R is a set of **relationship instances .**

Informally, each relationship instance in R is an association of entities, where the association includes exactly one entity from each participating entity type. Each such relationship instance represents the fact that the entities participating in are related in some way in the corresponding miniworld situation.

For example, consider a relationship type WORKS_FOR between the two entity types EMPLOYEE and DEPARTMENT, which associates each employee with the department the employee works for. Each relationship instance in the relationship set WORKS_FOR associates one employee entity and one department entity.
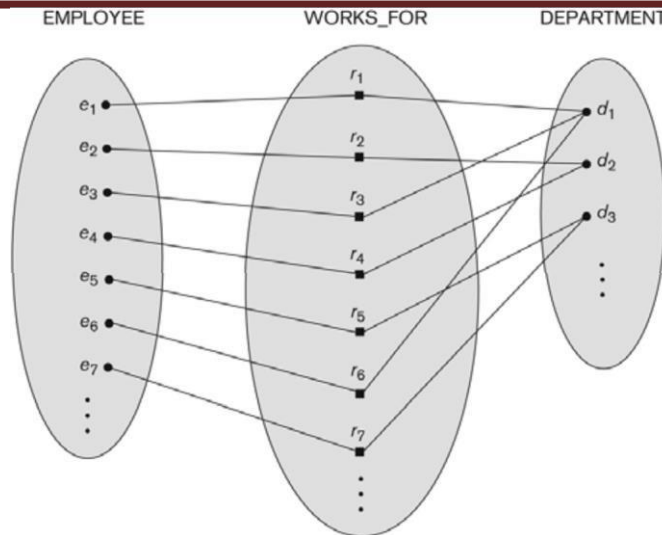
Figure 3.9(
Some instances in the
WORKS_FOR relationship
set, which represents a rela-
tionship type WORKS_FOR
between EMPLOYEE and
DEPARTMENT.

The above figure illustrates this example, where each relationship instance is shown connected to the employee and department entities that participate in .*E*mployees e1, e3, and e6 work for department d1; e2 and e4 work for d2; and e5 and e7 work for d3.

In ER diagrams, relationship types are displayed as diamond-shaped boxes, which are connected by straight lines to the rectangular boxes representing the participating entity types. The relationship name is displayed in the diamond-shaped box .

4.2 Degree of a Relationship Type

The **degree** of a relationship type is the number of participating entity types. Hence, the WORKS_FOR relationship is of degree two. A relationship type of degree two is called **binary,** and one of degree three is called **ternary.**

An example of a ternary relationship is SUPPLY, shown in above Figure . where each relationship instance associates three entities—a supplier s, a part p, and a project j—whenever s supplies part p to project j. Relationships can generally be of any degree, but the ones most common are binary relationships. Higher-degree relationships are generally more complex than binary relationships.

Relationships as Attributes

It is sometimes convenient to think of a relationship type in terms of attributes. One can think of an attribute called Department of the EMPLOYEE entity type whose value for each employee entity is (a reference to) the *department entity* that the employee works for.

Role Names and Recursive Relationships

Each entity type that participates in a relationship type plays a particular **role** in the relationship. The **role name** signifies the role that a participating entity from the entity type plays in each relationship instance, and helps to explain what the relationship means.

For example, in the WORKS_FOR relationship type, EMPLOYEE plays the role of *employee* or *worker* and DEPARTMENT plays the role of *department* or *employer*.

Role names are not technically necessary in relationship types where all the participating entity types are distinct, since each entity type name can be used as the role name. However, in some cases the *same* entity type participates more than once in a relationship type in *different roles.* In such cases the role name becomes essential for distinguishing the meaning of each participation. Such relationship types are called **recursive relationships.**
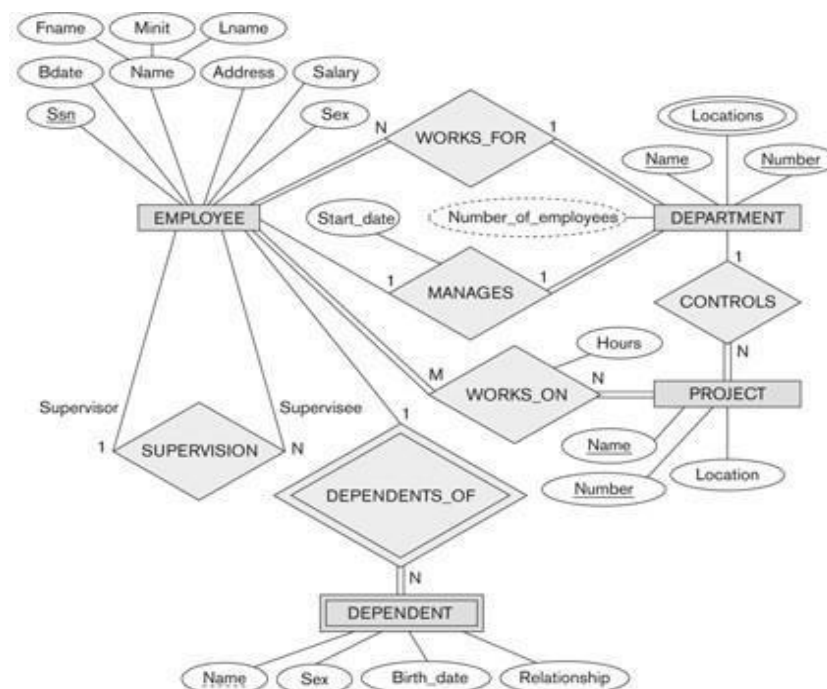
Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation

The above figure shows an example. The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity type. Hence, the EMPLOYEE entity type *participates twice* in SUPERVISION: once in the role of *supervisor* (or boss), and once in the role of *supervisee* (or subordinate).

Figure 3.11
A recursive relationship SUPERVISION between EMPLOYEE in the *supervisor* role (1) and EMPLOYEE in the *subordinate* role (2).
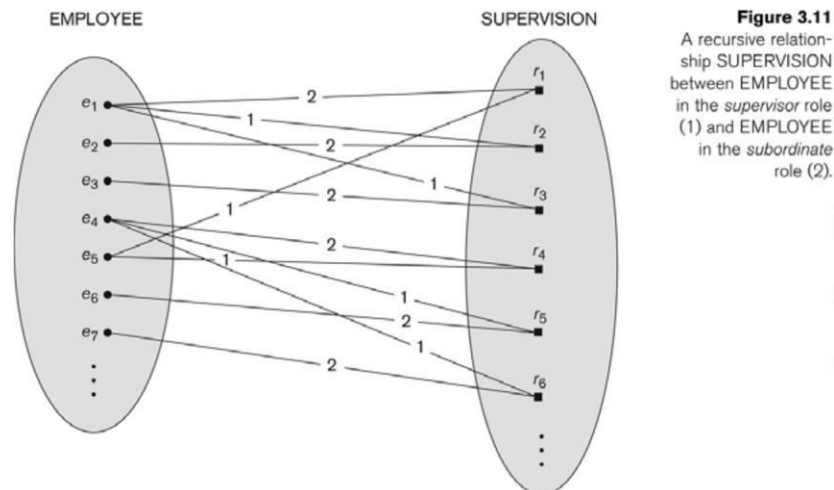
Each relationship instance in SUPERVISION associates two employee entities ejand ek, one of which plays the role of supervisor and the other the role of supervisee.

In the above figure the lines marked "1" represent the supervisor role, and those marked "2" represent the supervisee role; hence, e1 supervises e2 and e3; e4 supervises e6 and e7; and e5 supervises e1 and e4.

4.3 Constraints on Relationship Types

Cardinality Ratios for Binary Relationships Participation Constraints and Existence Dependencies.Relationship types usually have certain constraints that limit the possible combinations of entities that may participate in the corresponding relationship set. These constraints are determined from the mini world situation that the relationships represent.

For example, if the company has a rule that each employee must work for exactly one department, then we would like to describe this constraint in the schema. We can distinguish two main types of relationship constraints: *cardinality ratio* and *participation.*

Cardinality Ratios for Binary Relationships

cardinality ratio :

      cardinality ratio for a binary relationship specifies the number of relationship instances that an entity can participate in.

      For example, in the WORKS_FOR binary relationship type, DEPARTMENT:EMPLOYEEis of cardinality ratio 1:N, meaning that each department can be related to numerous employees but an employee can be related to only one department.
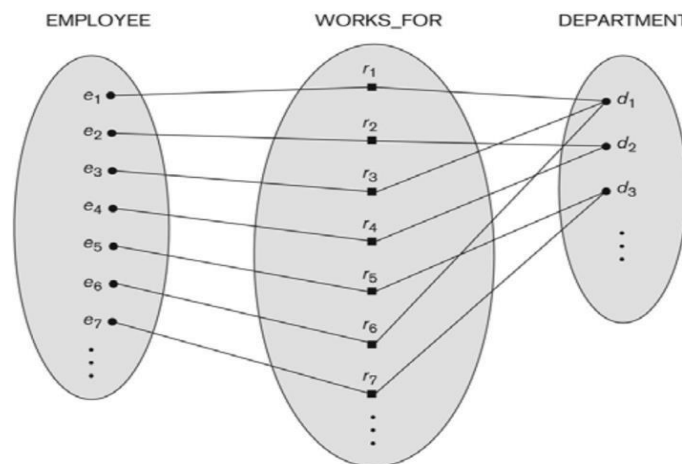
Figure 3.9
Some instances in the
WORKS_FOR relationship
set, which represents a rela-
tionship type WORKS_FOR
between EMPLOYEE and
DEPARTMENT.

      The possible cardinality ratios for binary relationship types are 1:1, 1:N, N:1, and M:N.

      An example of a 1:1 binary relationship is MANAGES which relates a department entity to the employee who manages that department. This represents the miniworld constraints that an employee can manage only one department and that a department has only one manager.

The relationship type WORKS_ON is of cardinality ratio M:N, because the miniworld rule is that an employee can work on several projects and a project can have several employees.
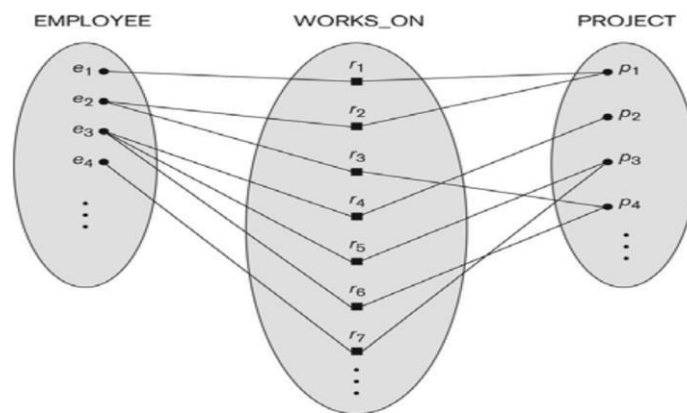
Figure 3.13
An M:N relationship,
WORKS_ON.

Cardinality ratios for binary relationships are displayed on ER diagrams by displaying 1, M, and N on the diamonds .

Participation Constraints and Existence Dependencies

The **participation constraint** specifies whether the existence of an entity depends on its being related to another entity via the relationship type. There are two types of participation constraints—**total** and **partial**—which we illustrate by example.

a) If a company policy states that *every* employee must work for a department, then an employee entity can exist only if it participates in a WORKS_FOR relationship instance .Thus, the participation of EMPLOYEE in WORKS_FOR is called **total participation,** meaning that every entity in "the total set" of employee entities must be related to a department entity via WORKS_FOR. Total participation is also called **existence dependency.**

b) we do not expect every employee to manage a department, so the participation of EMPLOYEE in the MANAGES relationship type is **partial,** meaning that *some* or "part of the set of" employee entities are related to a department entity via MANAGES, but not necessarily all.

structural constraints

cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type.

In ER diagrams, total participation is displayed as a *double line* connecting the participating entity type to the relationship, whereas partial participation is represented by a *single line* .

Attributes of Relationship Types

Relationship types can also have attributes, similar to those of entity types. For example, to record the number of hours per week that an employee works on a particular project, we can include an attribute **Hours** for the WORKS_ON relationship type .

Another example is to include the date on which a manager started managing a department via an attribute **StartDate**for the MANAGES relationship type .

Notice that attributes of 1:1 or 1:N relationship types can be migrated to one of the participating entity types. For example, the StartDate attribute for the MANAGES relationship can be an attribute of either EMPLOYEE or DEPARTMENT—although conceptually it belongs to MANAGES. This is because MANAGES is a 1:1 relationship, so every department or employee entity participates in *at most one* relationship instance. Hence, the value of the StartDate attribute can be determined separately, either by the participating department entity or by the participating employee (manager) entity.

For a 1:N relationship type, a relationship attribute can be migrated *only* to the entity type at the N-side of the relationship. For example if the WORKS_FOR relationship also has an attribute StartDate that indicates when an employee started working for a department, this attribute can be included as an attribute of EMPLOYEE. This is because each employee entity participates in at most one relationship instance in WORKS_FOR. In both 1:1 and 1:N relationship types, the decision as to where a relationship attribute should be placed—as a relationship type attribute or as an attribute of a participating entity type—is determined subjectively by the schema designer.

For M:N relationship types, some attributes may be determined by the *combination of participating entities* in a relationship instance, not by any single entity. Such attributes *must be specified as relationship attributes.* An example is the Hours attribute of the M:N relationship WORKS_ON . the number of hours an employee works on a project is determined by an employee-project combination and not separately by either entity.

5 Weak Entity Types

Entity types that do not have key attributes of their own are called **weak entity types.** In contrast, **regular entity types** that do have a key attribute are sometimes called **strong entity types.** Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with some of their attribute values. We call this other entity type the **identifying** or **owner entity type** and we call the relationship type that relates a weak entity type to its owner the **identifying relationship** of the weak entity type ..A weak entity type always has a *total participation constraint* (existence dependency) with respect to its identifying relationship, because a weak entity cannot be identified without an owner entity.

Consider the entity type DEPENDENT, related to EMPLOYEE, which is used to keep track of the dependents of each employee via a 1:N relationship .The attributes of DEPENDENT are Name (the first name of the dependent), BirthDate, Sex, and Relationship (to the employee). Two dependents of *two distinct employees* may, by chance, have the same values for Name, BirthDate, Sex, and Relationship, but they are still distinct entities. They are identified as distinct entities only after determining the *particular employee entity* to which each dependent is related. Each employee entity is said to **own** the dependent entities that are related to it.

A weak entity type normally has a **partial key,** which is the set of attributes that can uniquely identify weak entities that are *related to the same owner entity* .In our example, if we assume that no two dependents of the same employee ever have the same first name, the attribute Name of DEPENDENT is the partial key. In the worst case, a composite attribute of *all the weak entity's attributes* will be the partial key.

In ER diagrams, both a weak entity type and its identifying relationship are distinguished by surrounding their boxes and diamonds with *double lines*. The partial key attribute is underlined with a dashed or dotted line.

6 Refining the ER Design for the COMPANY Database

In our example, we specify the following relationship types:

1. MANAGES, a 1:1 relationship type between EMPLOYEE and DEPARTMENT. EMPLOYEE participation is partial. DEPARTMENT participation is not clear from the requirements. We question the users, who say that a department must have a manager at

all times, which implies total participation .The attribute StartDate is assigned to this relationship type.
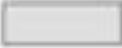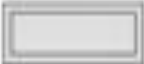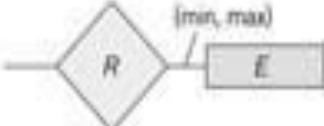
2. WORKS_FOR, a 1:N relationship type between DEPARTMENT and EMPLOYEE. Both participations are total.

3. CONTROLS, a 1:N relationship type between DEPARTMENT and PROJECT. The participation of PROJECT is total, whereas that of DEPARTMENT is determined to be partial, after consultation with the users.

4. SUPERVISION, a 1:N relationship type between EMPLOYEE (in the supervisor role) and EMPLOYEE (in the supervisee role). Both participations are determined to be partial, after the users indicate that not every employee is a supervisor and not every employee has a supervisor.

5. WORKS_ON, determined to be an M:N relationship type with attribute Hours, after the users indicate that a project can have several employees working on it. Both participations are determined to be total.

6. DEPENDENTS_OF, a 1:N relationship type between EMPLOYEE and DEPENDENT, which is also the identifying relationship for the weak entity type DEPENDENT. The participation of EMPLOYEE is partial, whereas that of DEPENDENT is total.

Summary of Notation for ER Diagrams

In ER diagrams the emphasis is on representing the schemas rather than the instances. This is more useful because a database schema changes rarely, whereas the extension changes frequently. In addition, the schema is usually easier to display than the extension of a database, because it is much smaller.

Figure 3.14
Summary of the
notation for ER
diagrams.

Symbol	Meaning

Entity

Weak Entity

Relationship

Indentifying Relationship

Attribute

Key Attribute

Multivalued Attribute

Composite Attribute

Derived Attribute

E_1 — R — E_2 Total Participation of E_2 in R

E_1 —1— R —N— E_2 Cardinality Ratio 1 : N for E_1 : E_2 in R

(min, max)
R — E   Structural Constraint (min, max) on Participation of E in R

The below Figure displays the COMPANY **ER database schema** as an ER diagram.
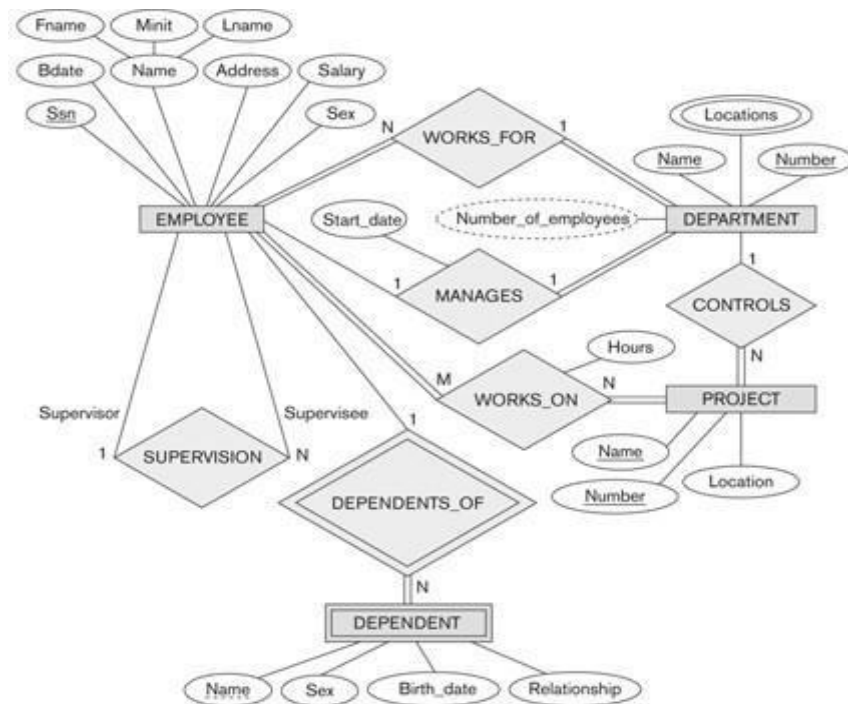

Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation

Entity types such as EMPLOYEE, DEPARTMENT, and PROJECT are shown in rectangular boxes. Relationship types such as WORKS_FOR, MANAGES, CONTROLS, and WORKS_ON are shown in diamond-shaped boxes attached to the participating entity types with straight lines. Attributes are shown in ovals, and each attribute is attached by a straight line to its entity type or relationship type. Component attributes of a composite attribute are attached to the oval representing the composite attribute, as illustrated by the Name attribute of EMPLOYEE. Multivalued attributes are shown in double ovals, as illustrated by the Locations attribute of DEPARTMENT. Key attributes have their names underlined. Derived attributes are shown in dotted ovals, as illustrated by the NumberOfEmployees attribute of DEPARTMENT.

Weak entity types are distinguished by being placed in double rectangles and by having their identifying relationship placed in double diamonds, as illustrated by the DEPENDENT entity type and the DEPENDENTS_OF identifying relationship type. The partial key of the weak entity type is underlined with a dotted line.

In Figure the cardinality ratio of each *binary* relationship type is specified by attaching a 1, M, or N on each participating edge. The cardinality ratio of DEPARTMENT: EMPLOYEE in

MANAGES is 1:1, whereas it is 1:N for DEPARTMENT:EMPLOYEE in WORKS_FOR, and it is M:N for WORKS_ON. The participation constraint is specified by a single line for partial participation and by double lines for total participation (existence dependency).

In Figure we show the role names for the SUPERVISION relationship type because the EMPLOYEE entity type plays both roles in that relationship. Notice that the cardinality is 1:N from supervisor to supervisee because, on the one hand, each employee in the role of supervisee has at most one direct supervisor, whereas an employee in the role of supervisor can supervise zero or more employees.