Module 01

Chapter 01: Databases and Database Users

Databases and database systems are an essential component of life in modern society: most of us encounter several activities every day 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 purchase something online—such as a book, toy, or computer—chances are that our activities will involve someone or some computer program accessing a database. Even purchasing items at a supermarket often automatically updates the database that holds the inventory of grocery items.

These interactions are examples of what we may call traditional database applications, in which most of the information that is stored and accessed is either textual or numeric. The proliferation of social media Web sites, such as Facebook, Twitter, and Flickr, among many others, has required the creation of huge databases that store nontraditional data, such as posts, tweets, images, and video clips. New types of database systems, often referred to as big data storage systems, or NOSQL systems, have been created to manage data for social media applications.

1.1 Introduction

Databases and database technology have had a major impact on the growing use of computers.

A **database** is a collection of related data.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.

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 management system (DBMS) is a computerized system that enables users to create and maintain a database.

“The DBMS is a general-purpose software system that facilitates the processes of defining, constructing, manipulating, and sharing databases among various users and applications.”

- Defining a database involves specifying the data types, structures, and constraints of the data to be stored in the database. The database definition or descriptive information is also stored by the DBMS in the form of a database catalog or dictionary; it is called meta-data.

- Constructing the database is the process of storing the data on some storage medium that is controlled by the DBMS.
- Manipulating a database includes functions such as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data.
- Sharing a database allows multiple users and programs to access the database simultaneously.

To complete our initial definitions, we will call the database and DBMS oftware together a **database system**. Figure 1.1 illustrates some of the concepts we have discussed so far.

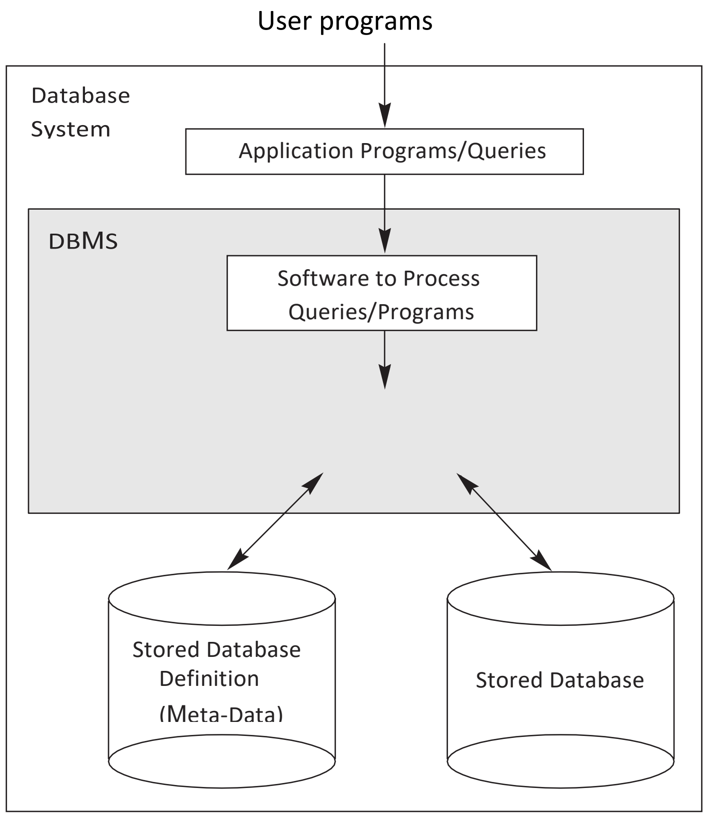

User programs
Database System
Application Programs/Queries
DBMS
Software to Process Queries/Programs
Stored Database Definition (Meta-Data)
Stored Database

Figure 1.1 : A simplified database system environment.

1.2 An Example

Let us consider a simple example that most readers may be familiar with: a UNIVERSITY database for maintaining information concerning students, courses, and grades in a university environment. Figure 1.2 shows the database structure and a few sample data records. The database is organized as five files, each of which stores data records of the same type.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.

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

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	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	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 Structure of STUDENT database.

1.3 Characteristics of the Database Approach

A number of characteristics distinguish the database approach from the much older approach of writing customized programs to access data stored in files. In traditional file processing, each user defines and implements the files needed for a specific software application as part of programming the application. For example, one user, the grade reporting office, may keep files on students and their grades. Programs to print a student's transcript and to enter new grades are implemented as part of the application. 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 up-to-date data.

In the database approach, a single repository maintains data that is defined once and then accessed by various users repeatedly through queries, transactions, and application programs. The main characteristics of the database approach versus the file-processing approach are the following:

■ Self-describing nature of a database system

■ Insulation between programs and data, and data abstraction

■ Support of multiple views of the data

■ Sharing of data and multiuser transaction processing

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 DBMS 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.

The catalog is used by the DBMS software and also by database users who need information about the database structure.

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. Whereas file-processing software can access only specific databases, DBMS software can access diverse databases by extracting the database definitions from the catalog and 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 application programs, so any changes to the structure of a file may require changing all programs that access that 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 some types of database systems, such as object-oriented and object-relational systems users can define operations on data as part of the database definitions. An operation (also called a function or method) 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. 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.

1.3.3 Support of Multiple Views of the Data

A database typically has many types of users, each of whom may require a different perspective or view of the database. 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 distinct applications must provide facilities for defining multiple views.

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. A fundamental role of multiuser DBMS software is to ensure that concurrent transactions operate correctly and efficiently.

The concept of a transaction has become central to many database applications. A transaction is an executing program or process that includes one or more database accesses, such as reading or updating of database records. Each transaction is supposed to execute a logically correct database access if executed in its entirety without interference from other transactions. The DBMS must enforce several transaction properties. The isolation property ensures that each transaction appears to execute in isolation from other transactions, even though hundreds of transactions may be executing concurrently. The atomicity property ensures that either all the database operations in a transaction are executed or none are.

1.4 Actors on the Scene

In large organizations, many people are involved in the design, use, and maintenance of a large database with hundreds or thousands of users.

1.4.1 Database Administrators

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, coordinating and monitoring its use, and acquiring software and hardware resources as needed. The DBA is accountable for problems such as security breaches and poor system response time. In large organizations, the DBA is assisted by a staff that carries out these functions.

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 create a design that meets these requirements. Database designers typically interact with each potential group of users and develop views of the database that meet the data and processing requirements of these groups. Each view is then analyzed and integrated with the views of other user groups. The final database design must be capable of supporting the requirements of all user groups.

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 interface 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. Many of these tasks are now available as mobile apps for use with mobile devices. The tasks that such users perform are varied. A few examples are:
    - Bank customers and tellers check account balances and post withdrawals and deposits.
    - Reservation agents or customers for airlines, hotels, and car rental companies check availability for a given request and make reservations.

    - Employees at receiving stations for shipping companies enter package identifications via bar codes and descriptive information through buttons to update a central database of received and in-transit packages.
    - Social media users post and read items on social media Web sites.
- Sophisticated end users include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS in order to implement their own applications to meet their complex requirements.
- Standalone users maintain personal databases by using ready-made program packages that provide easy-to-use menu-based or graphics-based interfaces. An example is the user of a financial software package that stores a variety of personal financial data.

1.4.4 System Analysts and Application Programmers

System analysts determine the requirements of end users, especially naive and parametric end users, and develop specifications for standard 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—commonly referred to as software developers or 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 content itself. We call them the workers behind the scene, and they include the following categories:

■ DBMS system designers and implementers design and implement the DBMS modules and interfaces as a software package. A DBMS is a very complex software system that consists of many components, or modules, including modules for implementing the catalog, query language processing, interface processing, accessing and buffering data, controlling concurrency, and handling data recovery and security. The DBMS must interface with other system software, such as the operating system and compilers for various programming languages.

■ Tool developers design and implement tools—the software packages that facilitate database modelling and design, database system design, and improved 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. In many cases, independent software vendors develop and market these tools.

■ Operators and maintenance personnel (system administration personnel) are responsible for the actual running and maintenance of the hardware and software environment for the database system.

1.6 Advantages of Using the DBMS Approach

In this section we discuss some additional advantages of using a DBMS and the capabilities that a good DBMS should possess.

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 of Section 1.2; here, 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 keeps data on registration and related billing information, whereas the registration office keeps track of student courses and grades. Other 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. Ideally, 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 is known as data normalization, and it ensures consistency and saves storage space.

1.6.2 Restricting Unauthorized Access

When multiple users share a large database, it is likely that most users will not be authorized to access all information in the database. For example, financial data such as salaries and bonuses is often considered confidential, and only authorized persons are allowed to access such data. In addition, some users may only be permitted to retrieve data, whereas others are allowed to retrieve and update. Hence, the type of access operation—retrieval or update—must also be controlled.

1.6.3 Providing Persistent Storage for Program Objects

Databases can be used to provide persistent storage for program objects and data structures. Programming languages typically have complex data structures, such as structs or class definitions in C++ or Java. The values of program variables or objects are discarded once a program terminates, unless the programmer explicitly stores them in permanent files, which often involves converting these complex structures into a format suitable for file storage. When the need arises to read this data once more, the programmer must convert from the file format to the program variable or object structure. Object-oriented database systems are compatible with programming languages such as C++ and Java, and the DBMS software automatically performs any necessary conversions.

Hence, a complex object in C++ can be stored permanently in an object-oriented DBMS. Such an object is said to be persistent.

1.6.4 Providing Storage Structures and Search Techniques for Efficient Query Processing

Database systems must provide capabilities for efficiently executing queries and updates. Because the database is typically stored on disk, the DBMS must provide specialized data structures and search techniques to speed up disk search for the desired records. Auxiliary files called indexes are often used for this purpose. Therefore, the DBMS often has a buffering or caching module that maintains parts of the database in main memory buffers.

The query processing and optimization module of the DBMS is responsible for choosing an efficient query execution plan for each query based on the existing storage structures. The choice of which indexes to create and maintain is part of physical database design and tuning, which is one of the responsibilities of the DBA staff.

1.6.5 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 transaction, the recovery subsystem is responsible for making sure that the database is restored to the state it was in before the transaction started executing. Disk backup is also necessary in case of a catastrophic disk failure.

1.6.6 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 apps for mobile users, 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 standalone users. Both forms-style interfaces and menu-driven interfaces are commonly known as graphical user interfaces (GUIs).

1.6.7 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, to define new relationships as they arise, and to retrieve and update related data easily and efficiently.

1.6.8 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. A more complex type of constraint that frequently occurs involves specifying that a record in one file must be related to records in other files. We can specify that every section record must be related to a course record. This is known as a referential integrity constraint. Another type of constraint specifies uniqueness on data item values, such as every course record must have a unique value for

Course_number. This is known as a key or uniqueness constraint. These constraints are derived from the meaning or semantics of the data and of the miniworld it represents. It is the responsibility of the database designers to identify integrity constraints during database design.

1.6.9 Permitting Inference and Actions Using Rules and Triggers

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. In today's relational database systems, it is possible to associate triggers with tables. A trigger is a form of a rule activated by updates to the table, which results in performing some additional operations to some other tables, sending messages, and so on. More involved procedures to enforce rules are popularly called stored procedures; they become a part of the overall database definition and are invoked appropriately when certain conditions are met. More powerful functionality is provided by active database systems, which provide active rules that can automatically initiate actions when certain events and conditions occur.

1.6.10 Additional Implications of Using the Database Approach

Few additional implications of using the database approach that can benefit most organizations.

Potential for Enforcing Standards: The database approach permits the DBA to define and enforce standards among database users in a large organization. This facilitates communication and cooperation among various departments, projects, and users within the organization. Standards can be defined for names and formats of data elements, display formats, report structures, terminology, and so on.

Reduced Application Development Time: Designing and implementing a large multiuser database from scratch may take more time than writing a single specialized file application. However, once a database is up and running, substantially less time is generally required to create new applications using DBMS facilities. Development time using a DBMS is estimated to be one sixth to one-fourth of that for a file system.

Flexibility: It may be necessary to change the structure of a database as requirements change. Modern DBMSs allow certain types of evolutionary 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, and it is made possible by the concurrency control and recovery subsystems of a DBMS.

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 as well as redundancies among applications. This reduces overall costs of operation and management.

1.7 A Brief History of Database Applications

1.7.1 Early Database Applications Using Hierarchical and Network Systems

Many early database applications maintained records in large organizations such as corporations, universities, hospitals, and banks. In many of these applications, there were large numbers of records of similar structure. One of the main problems with early database systems was the intermixing of conceptual relationships with the physical storage and placement of records on disk. Hence, these systems did not provide sufficient data abstraction and program-data independence capabilities. It was also laborious to reorganize the database when changes were made to the application's requirements.

Another shortcoming of early systems was that they provided only programming language interfaces. This made it time-consuming and expensive to implement new queries and transactions, since new programs had to be written, tested, and debugged. The main types of early systems were based on three main paradigms: hierarchical systems, network model–based systems, and inverted file systems.

1.7.2 Providing Data Abstraction and Application Flexibility with Relational Databases

Relational databases were originally proposed to separate the physical storage of data from its conceptual representation and to provide a mathematical foundation for data representation and querying. The relational data model also introduced high-level query languages that provided an alternative to programming language interfaces, making it much faster to write new queries.In these systems, data abstraction and program-data independence were much improved when compared to earlier systems.

With the development of new storage and indexing techniques and better query processing and optimization, their performance improved. Eventually, relational databases became the dominant type of database system for traditional database applications. Relational databases now exist on almost all types of computers, from small personal computers to large servers.

1.7.3 Object-Oriented Applications and the Need for More Complex Databases

The emergence of object-oriented programming languages in the 1980s and the need to store and share complex, structured objects led to the development of object-oriented databases (OODBs). They also incorporated many of the useful object-oriented paradigms, such as abstract data types, encapsulation of operations, inheritance, and object identity. However, the complexity of the model and the lack of an early standard contributed to their limited use. They are now mainly used in specialized applications, such as engineering design, multimedia publishing, and manufacturing systems.

In addition, many object-oriented concepts were incorporated into the newer versions of relational DBMSs, leading to object-relational database management systems, known as ORDBMSs.

1.7.4 Interchanging Data on the Web for E-Commerce Using XML

Starting in the 1990s, electronic commerce (e-commerce) emerged as a major application on the Web. Much of the critical information on e-commerce Web pages is dynamically extracted data from

DBMSs, such as flight information, product prices, and product availability. A variety of techniques were developed to allow the interchange of dynamically extracted data on the Web for display on Web pages. The eXtended Markup Language (XML) is one standard for interchanging data among various types of databases and Web pages. XML combines concepts from the models used in document systems with database modelling concepts.

1.7.5 Extending Database Capabilities for New Applications

Database systems now offer extensions to better support the specialized requirements for some of these applications. The following are some examples of these applications:

- Scientific applications that store large amounts of data resulting from scientific experiments in areas such as high-energy physics, the mapping of the human genome, and the discovery of protein structures.
- Storage and retrieval of images, including scanned news or personal photographs, satellite photographic images, and images from medical procedures such as x-rays and MRI (magnetic resonance imaging) tests.
- Storage and retrieval of videos, such as movies, and video clips from news or personal digital cameras.
- Data mining applications that analyze large amounts of data to search for the occurrences of specific patterns or relationships, and for identifying unusual patterns in areas such as credit card fraud detection.
- Spatial applications that store and analyze spatial locations of data, such as weather information, maps used in geographical information systems, and automobile navigational systems.
- Time series applications that store information such as economic data at regular points in time, such as daily sales and monthly gross national product figures.

It was quickly apparent that basic relational systems were not very suitable for many of these applications, usually for one or more of the following reasons:

■ More complex data structures were needed for modeling the application than the simple relational representation.

■ New data types were needed in addition to the basic numeric and character string types.

■ New operations and query language constructs were necessary to manipulate the new data types.

■ New storage and indexing structures were needed for efficient searching on the new data types.

1.7.6 Emergence of Big Data Storage Systems and NOSQL Databases

In the first decade of the twenty-first century, the proliferation of applications and platforms such as social media Web sites, large e-commerce companies, Web search indexes, and cloud storage/backup led to a surge in the amount of data stored on large databases and massive servers. New types of database systems were necessary to manage these huge databases—systems that would provide fast search and retrieval as well as reliable and safe storage of non-traditional types of data, such as social media posts and tweets. Some of the requirements of these new systems were not compatible with SQL relational DBMSs. The term NOSQL is generally interpreted as Not Only SQL, meaning that in systems than manage large amounts of data, some of the data is stored using SQL systems, whereas other data would be stored using NOSQL, depending on the application requirements.

Chapter 02: Database System Concepts and Architecture

In basic client/server DBMS architecture, the system functionality is distributed between two types of modules.1 A **client module** is typically designed so that it will run on a mobile device, user workstation, or personal computer (PC). The client module handles user interaction and provides the user-friendly interfaces such as apps for mobile devices, or forms- or menu based GUIs for PCs. The other kind of module, called a **server module**, typically handles data storage, access, search, and other functions.

2.1 Data Models, Schemas, and Instances

One fundamental characteristic of the database approach is that it provides some level of data abstraction. Data abstraction generally refers to the suppression of details of data organization and storage, and the highlighting of the essential features for an improved understanding of data. . One of the main characteristics of the database approach is to support data abstraction so that different users can perceive data at their preferred level of detail.

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 apply to the data.

2.1.1 Categories of Data Models

Many data models have been proposed, which we can categorize according to the types of concepts they use to describe the database structure.

- High-level or conceptual data models
- Low-level or physical data models
- Representational (or implementation) data models

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

Conceptual data models use concepts such as entities, attributes, and relationships. An entity represents a real-world object or concept, such as an employee or a project from the miniworld 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 association among the entities.

Example : **Entity–Relationship model** is the most popular high-level conceptual data model.

Low-level or physical data models provide concepts that describe the details of how data is stored on the computer storage media, typically magnetic disks.

Representational (or implementation) data models, which provide concepts that may be easily understood by end users but that are not too far removed from the way data is organized in computer storage. Representational data models hide many details of data storage on disk but can be implemented on a computer system directly.

Representational or implementation data models are the models used most frequently in traditional commercial DBMSs. These include the widely used relational data model, as well as the so-called legacy data models—**the network and hierarchical models**.

Representational data models represent data by using record structures and hence are sometimes called **record-based data models**

Another class of data models is known as **self-describing data models**. The data storage in systems based on these models combines the description of the data with the data values themselves. In traditional DBMSs, the description (schema) is separated from the data.

Example: **XML, key-value stores** and **NOSQL systems** that were recently created for managing big data.

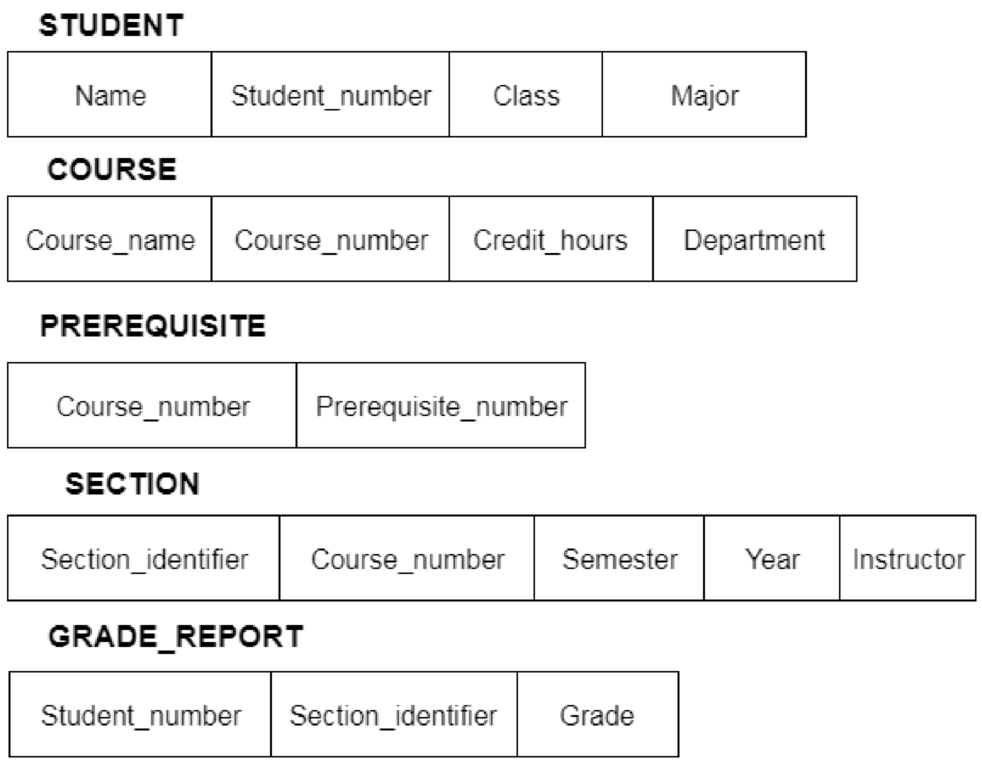
2.1.2 Schemas, Instances, and Database State

Database schema

The **description of a database** is called the **database schema**, which is specified during database design and is not expected to change frequently.

Schema diagram

A displayed schema is called a schema diagram.



STUDENT

Name	Student_number	Class	Major

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

Figure 2.1: Schema diagram for Student database.

We call each object in the schema—such as STUDENT or COURSE—a **schema construct**.

Database state

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.

Valid state— a state that satisfies the structure and constraints specified in the schema.

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 is called an **extension** of the schema.

2.2 Three-Schema Architecture and Data Independence

In this section we specify an architecture for database systems, called the three-schema architecture, that was proposed to help achieve and visualize the following three characteristics:

(1) Use of a catalog to store the database description so as to make it self-describing,

(2) Insulation of programs and data (program-data and program-operation independence), and

(3) Support of multiple user views.

2.2.1 The Three-Schema Architecture

The goal of the three-schema architecture is to separate the user applications from the physical database. In this architecture, schemas can be defined at the following three levels:

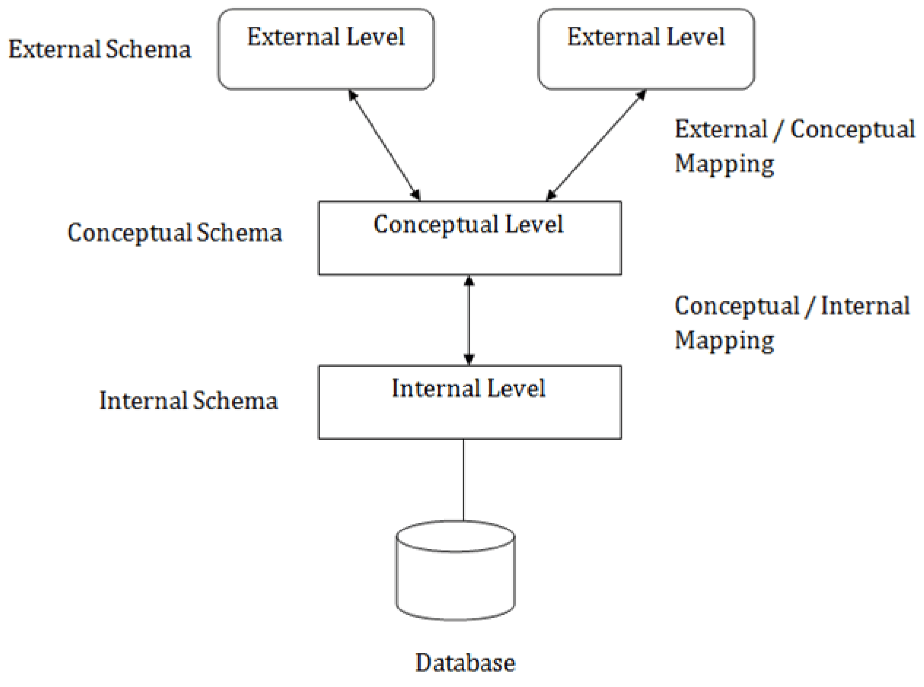


Figure 2.2: Three-schema architecture

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.
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.

The three-schema architecture is a convenient tool with which the user can visualize the schema levels in a database system. Most DBMSs do not separate the three levels completely and explicitly, but they support the three-schema architecture to some extent.

In the three-schema architecture, each user group refers to its own external schema. Hence, the 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 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**.

2.2.2 Data Independence

Data independence 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.
2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual schema. Hence, the external schemas need not be changed as well.

2.3 Database Languages and Interfaces

In this section we discuss the types of languages and interfaces provided by a DBMS and the user categories targeted by each interface.

2.3.1 DBMS Languages

Four languages:

- data definition language (DDL)
- Storage definition language (SDL)
- View definition language (VDL)
- Data manipulation language (DML)

In many DBMSs where no strict separation of levels is maintained, **data definition language (DDL)**, is used by the DBA and by database designers to define both schemas. In DBMSs where a clear separation is maintained between the conceptual and internal levels, the DDL is used to specify the conceptual schema and the **storage definition language** (SDL), is used to specify the internal schema.

For a true three-schema architecture, we would need a third language, the **view definition language** (VDL), to specify user views and their mappings to the conceptual schema.

Once the database schemas are compiled and the database is populated with data, users must have some means to manipulate the database. The DBMS provides a set of operations or a language called the **data manipulation language (DML)** for these purposes.

There are two main types of DMLs.

- A high-level or nonprocedural DML
- A lowlevel or procedural DML

A **high-level or nonprocedural DML** can be used on its own to specify complex database operations concisely.

A low-level or procedural DML must be embedded in a general-purpose programming language.

2.3.2 DBMS Interfaces

User-friendly interfaces provided by a DBMS includes the following:

1. **Menu-based Interfaces for Web Clients or Browsing:** These interfaces present the user with lists of options (called menus) that lead the user through the formulation of a request. Pull-down menus are a very popular technique in Web-based user interfaces.

2. **Apps for Mobile Devices:** These interfaces present mobile users with access to their data. For example, banking, reservations, and insurance companies, among many others, provide apps that allow users to access their data through a mobile phone or mobile device.

3. **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 can 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.

4. **Graphical User Interfaces**: A GUI typically displays a schema to the user in diagrammatic form. The user then can specify a query by manipulating the diagram.

5. **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, as well as a dictionary of

important words. The natural language interface refers to the words in its schema, as well as to the set of standard words in its dictionary, that are used to interpret the request. If the interpretation is successful, the interface generates a high-level query corresponding to the natural language request and submits it to the DBMS for processing; otherwise, a dialogue is started with the user to clarify the request.

6. **Keyword-based Database Search**. These are somewhat similar to Web search engines, which accept strings of natural language (like English or Spanish) words and match them with documents at specific sites or Web pages on the Web at large (for engines like Google or Ask). They use predefined indexes on words and use ranking functions to retrieve and present resulting documents in a decreasing degree of match.

7. **Speech Input and Output**: Applications with limited vocabularies, such as inquiries for telephone directory, flight arrival/departure, and credit card account information, are allowing speech for input and output to enable customers to access this information. The speech input is detected using a library of predefined words and used to set up the parameters that are supplied to the queries. For output, a similar conversion from text or numbers into speech takes place.

8. **Interfaces for Parametric Users**: Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. For example, a teller is able to use single function keys to invoke routine and repetitive transactions such as account deposits or withdrawals, or balance inquiries. Systems analysts and programmers design and implement a special interface for each known class of naive users.

9. **Interfaces for the DBA**: Most database systems contain privileged commands that can be used only by the DBA 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 2.3 illustrates, in a simplified form, the typical DBMS components. The figure is divided into two parts. The top part of the figure refers to the various users of the database environment and their interfaces. The lower part shows the internal modules of the DBMS responsible for storage of data and processing of transactions.

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 read/write. Many DBMSs have their own buffer management module to schedule disk read/write, because management of buffer storage has a considerable effect on performance. Reducing disk read/write improves performance considerably. 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.

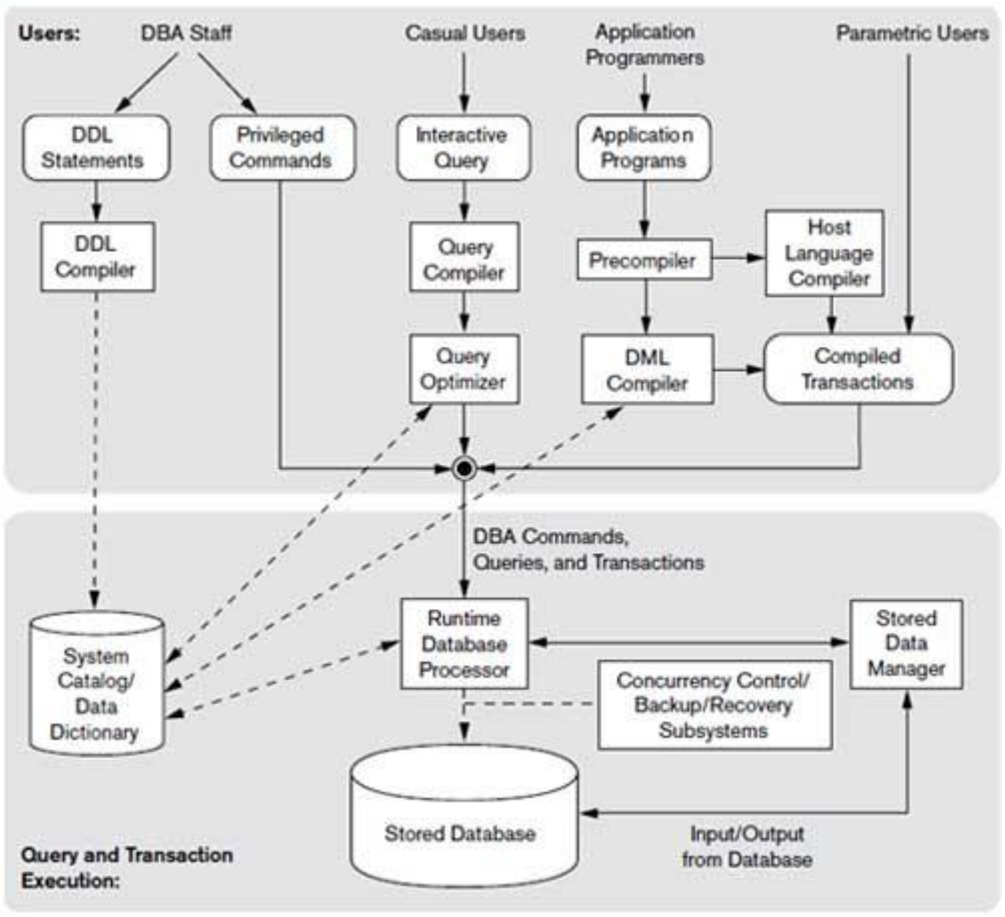


Figure 2.3: DBMS component modules.

Let us consider the top part of Figure 2.3 first. It shows interfaces for the DBA staff, casual users who work with interactive interfaces to formulate queries, application programmers who create programs using some host programming languages, and parametric users who do data entry work by supplying parameters to predefined transactions. The DBA staff works on defining the database and tuning it by making changes to its definition using the DDL and other privileged commands.

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 and sizes of files, names and data types of data items, storage details of each file, mapping information among schemas, and constraints.

Casual users and persons with occasional need for information from the database interact using the interactive query interface. We have not explicitly shown any menu-based or form-based or mobile interactions that are typically used to generate the interactive query automatically or to access **canned transactions**. These queries are parsed and validated for correctness of the query syntax, the names of files and data elements, and so on by a query compiler that compiles them into an internal form. This internal query is subjected to query optimization. Among other things, the query optimizer is concerned with the rearrangement and possible reordering of operations, elimination of redundancies, and use of efficient search algorithms during execution. It consults the **system catalog** for statistical and other physical information about the stored data and generates executable code that performs the necessary operations for the query and makes calls on the **runtime processor**.

Application programmers write programs in host languages such as Java, C, or C++ that are submitted to a precompiler. The **precompiler** 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**.

Canned transactions are executed repeatedly by parametric users via PCs or mobile apps; these users simply supply the parameters to the transactions. Each execution is considered to be a separate transaction. An example is a bank payment transaction where the account number, payee, and amount may be supplied as parameters.

In the lower part of Figure 2.3, the runtime database processor executes

(1) the privileged commands,

(2) the executable query plans, and

(3) the canned transactions with runtime parameters.

It works with the system catalog and may update it with statistics. It also works with the stored data manager, which in turn uses basic operating system services for carrying out low-level input/output (read/write) operations between the disk and main memory. The runtime database processor handles other aspects of data transfer, such as management of buffers in the main memory. Some DBMSs have their own buffer management module whereas others depend on the OS for buffer management.

2.4.2 Database System Utilities

In addition to possessing the software modules, most DBMSs have database utilities that help the DBA manage the database system. Common utilities have the following types of functions:

■ **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, which then automatically reformats the data and stores it in the database.

■ **Backup**: A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape or other mass storage medium. The backup copy can be used to restore the database in case of catastrophic disk failure

■ **Database storage reorganization**: This utility can be used to reorganize a set of database files into different file organizations and create new access paths to improve performance.

■ **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 or whether to add or drop indexes to improve performance.

2.6 Classification of Database Management Systems

Several criteria can be used to classify DBMSs. They are,

- Data model
- Number of users
- Number of sites
- Cost
- Types of access path

1. Data model:

The first is the data model on which the DBMS is based. The main data model used in many current commercial DBMSs is the **relational data model**, and the systems based on this model are known as SQL systems. The object data model has been implemented in some commercial systems but has not had widespread use. Recently, so-called big data systems, also known as **key-value storage** systems and **NOSQL systems**, use various data models: **document-based, graph-based, column-based**, and **key-value data models**. Many legacy applications still run on database systems based on the **hierarchical and network data models**.

Some experimental DBMSs are based on the **XML** (eXtended Markup Language) model, which is a **tree-structured data model**. These have been called native XML DBMSs.

2. Number of users:

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 PCs.
- **Multiuser systems -** support concurrent multiple users.

3.Number of sites:

The 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 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. Big data systems are often massively distributed, with hundreds of sites. The data is often replicated on multiple sites so that failure of a site will not make some data unavailable. **Homogeneous** DDBMSs use the same DBMS software at all the sites, whereas **Heterogeneous** DDBMSs can use different DBMS software at each site.

4.Cost:

The fourth criterion is cost. It is difficult to propose a classification of DBMSs based on cost. Today we have open source (free) DBMS products like MySQL and PostgreSQL that are supported by third-party vendors with additional services.

5.Types of access path:

We can also classify a DBMS 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.

Let us briefly elaborate on the main criterion for classifying DBMSs: the **data model**.

The **relational data model** represents a database as a collection of tables, where each table can be stored as a separate file. Most relational databases use the high-level query language called SQL and support a limited form of user views.

The **object data model** defines a database in terms of objects, their properties, and their operations. Objects with the same structure and behavior belong to a class, and classes are organized into hierarchies (or acyclic graphs). The operations of each class are specified in terms of predefined procedures called methods. Relational DBMSs have been extending their models to incorporate object database concepts and other capabilities; these systems are referred to as **object-relational or extended relational systems**.

Big data systems are based on various data models, with the following four data models most common.

- The **key-value data model** associates a unique key with each value (which can be a record or object) and provides very fast access to a value given its key.
- **The document data model** is based on JSON (Java Script Object Notation) and stores the data as documents, which somewhat resemble complex objects.
- **The graph data model** stores objects as graph nodes and relationships among objects as directed graph edges.
- **The column-based data models** store the columns of rows clustered on disk pages for fast access and allow multiple versions of the data.

The **XML model** has emerged as a standard for exchanging data over the Web and has been used as a basis for implementing several prototype native XML systems. XML uses hierarchical tree structures. It combines database concepts with concepts from document representation models. Data is represented as elements; with the use of tags, data can be nested to create complex tree structures

Two older, historically important data models, now known as legacy data models, **are the network and hierarchical models**.

The network model represents data as record types and also represents a limited type of 1:N relationship, called a set type. A 1:N, or one-to-many, relationship relates one instance of a record to many record instances using some pointer linking mechanism in these models.

The hierarchical model represents data as hierarchical tree structures. Each hierarchy represents a number of related records. There is no standard language for the hierarchical model.

Chapter 03: Data Modelling Using the Entity– Relationship Model

Conceptual modelling is a very important phase in designing a successful database application. Generally, the term database application refers to a particular database and the associated programs that implement the database queries and updates. For example, a BANK database application that keeps track of customer accounts would include programs that implement database updates corresponding to customer deposits and withdrawals. These programs would provide user-friendly graphical user interfaces (GUIs) utilizing forms and menus for the end users of the application—the bank customers or bank tellers in this example.

Hence, a major part of the database application will require the design, implementation, and testing of these application programs. Traditionally, the design and testing of application programs has been considered to be part of software engineering rather than database design. In many software design tools, the database design methodologies and software engineering methodologies are intertwined since these activities are strongly related.

3.1 Using High-Level Conceptual Data Models for Database Design

Figure 3.1 shows a simplified overview 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, including both retrievals and updates. In software design, it is common to use data flow diagrams, sequence diagrams, scenarios, and other techniques to specify functional requirements.

Once 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. This approach enables database designers to concentrate on specifying the properties of the data, without being concerned with storage and implementation details, which makes it is easier to create a good conceptual database design.

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 (SQL) 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**; its result is a **database schema** in the implementation data model of the DBMS. Data model mapping is often automated or semi automated within the database design tools.

The last step is the **physical design phase**, during which the internal storage structures, file organizations, indexes, access paths, and physical design parameters 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.

A simplified diagram to illustrate the main phases of database design.

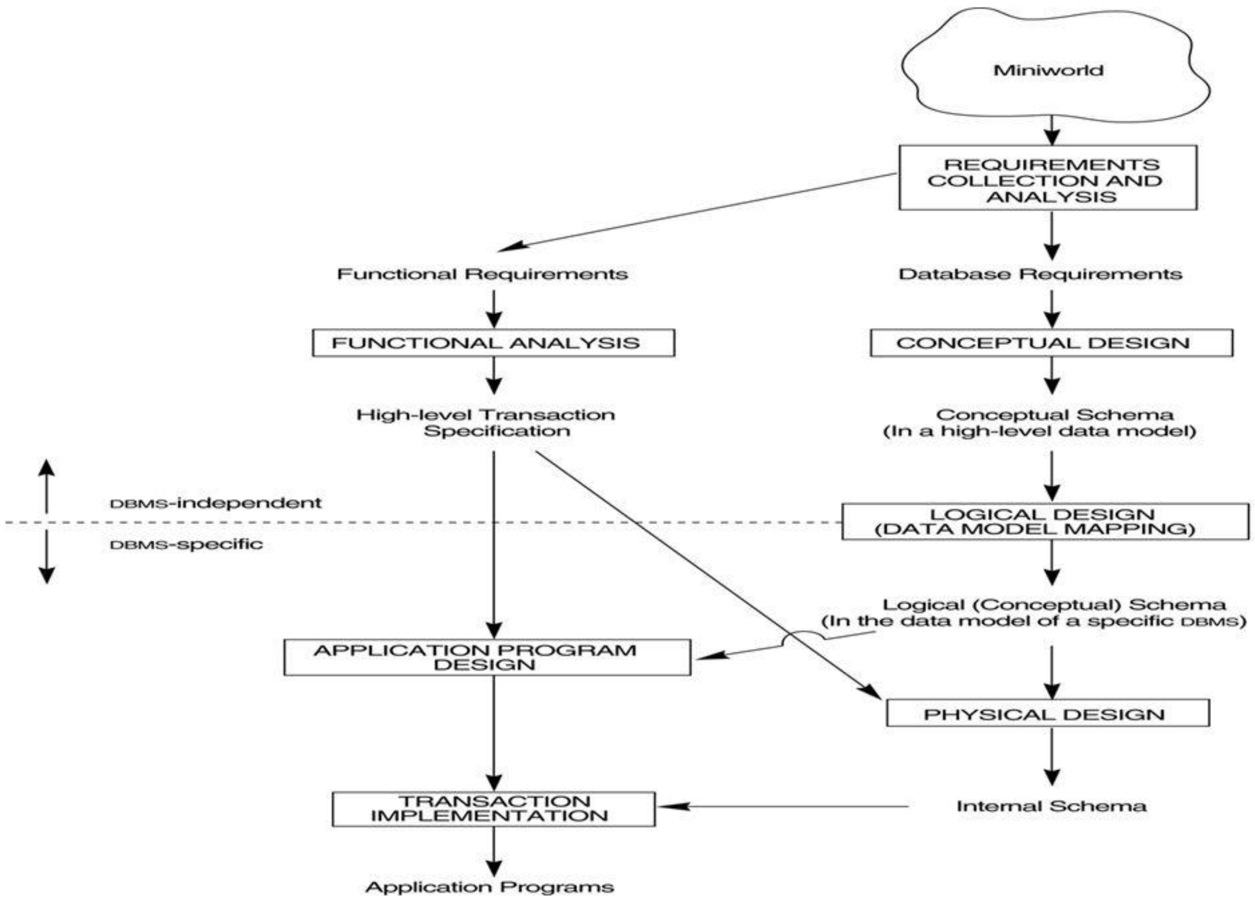

Miniworld
REQUIREMENTS COLLECTION AND ANALYSIS
Functional Requirements
Database Requirements
FUNCTIONAL ANALYSIS
CONCEPTUAL DESIGN
High-level Transaction Specification
Conceptual Schema (In a high-level data model)
DBMS-independent
DBMS-specific
LOGICAL DESIGN (DATA MODEL MAPPING)
Logical (Conceptual) Schema (In the data model of a specific DBMS)
APPLICATION PROGRAM DESIGN
PHYSICAL DESIGN
TRANSACTION IMPLEMENTATION
Internal Schema
Application Programs

Figure 3.1 Phases of Database Design.

3.2 A Sample Database Application

In this section we describe a sample database application, called COMPANY, which serves to illustrate the basic ER model concepts and their use in schema design. We list the data requirements for the database here, and then create its conceptual schema step-by-step as we introduce the modeling concepts of the ER model. The COMPANY database keeps track of a company's employees, departments, and projects. Suppose that after the requirements collection and analysis phase, the database designers provide the following description of the miniworld—the part of the company that will be represented in the database.

■ 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.

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

■ The database will store each employee’s name, Social Security number,2 address, salary, sex (gender), 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. It is required to keep track of the current number of hours per week that an employee works on each project, as well as the direct supervisor of each employee (who is another employee).

■ The database will keep track of the dependents of each employee for insurance purposes, including each dependent’s first name, sex, birth date, and relationship to the employee.

3.3 Entity Types, Entity Sets, Attributes, and Keys

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

3.3.1 Entities and Attributes

Entity

An entity is a thing or object in the real world with an independent existence. An entity may be an object with a physical existence (for example, a particular person, car, house, or employee) or it may be an object with a conceptual existence (for instance, a company, a job, or a university course).

Attribute

Each entity has attributes—the particular properties that describe it. The attribute values that describe each entity become a major part of the data stored in the database.

For example, an EMPLOYEE entity may be described by the employee’s name, age, address, salary, and job.

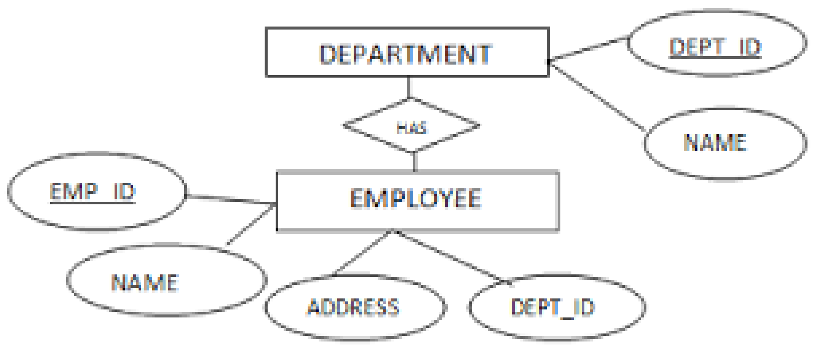


Figure 3.2: example for an entity.

Types of Attributes

Several types of attributes occur in the ER model:

- Simple Vs Composite
- Single-valued Vs Multi-valued
- Stored Vs Derived
- Null Attributes
- Complex Attributes

1. Composite versus Simple (Atomic) Attributes

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

Ex: age of a person

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

Ex: The Address attribute of the EMPLOYEE entity can be subdivided into Street_address, City, State, and Zip

Example of a composite attribute

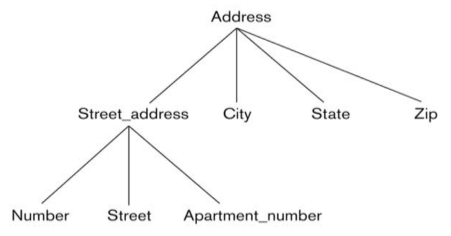


Figure 3.3 Composite Attributes

2. Single-Valued versus Multi-valued Attributes

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

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

An attribute which can have multiple values for a particular entity is called **Multi-valued attributes.**

Ex: In some cases an attribute can have a set of values for the same entity—for instance, a Colors attribute for a car, or a College_degrees attribute for a person. Cars with one color have a single value, whereas two-tone cars have two color values.

A multivalued attribute may have lower and upper bounds to constrain the number of values allowed for each individual entity. For example, the Colors attribute of a car may be restricted to have between one and two values, if we assume that a car can have two colors at most.

3. Stored versus Derived Attributes

In some cases, two (or more) attribute values are related—for example, the Age and Birth_date 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 Birth_date. The Age attribute is hence called a **derived attribute** and is said to be derivable from the Birth_date attribute, which is called a **stored attribute.**

4. NULL Values

In some cases, a particular entity may not have an applicable value for an attribute. For example, the Apartment_number 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. Similarly, a College_degrees attribute applies only to people with college degrees. For such situations, a special value called **NULL** is created.

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 number of 'John Smith'. The meaning of the former type of NULL is not applicable, whereas the meaning of the latter is unknown.

5. Complex Attributes

It is formed by nesting Composite and multi-valued attributes in an arbitrary way.

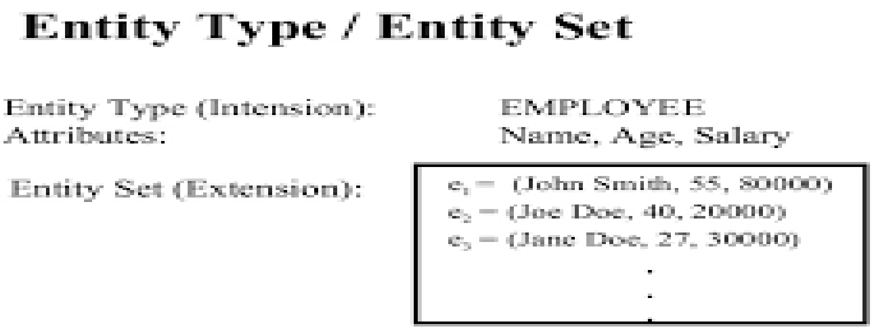
Ex:{Address_phone({Phone(Area_code,Phone_number)},

Address(Street_address (Number,Street,Apartment_number),City,State,Zip))}

3.3.2 Entity Types, Entity Sets, Keys, and Value Sets

Entity Type: 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.

Entity Set: The collection of all entities of a particular entity type in the database at any point in time is called an entity set or entity collection.



Entity Type / Entity Set

Entity Type (Intension): EMPLOYEE
Attributes: Name, Age, Salary

Entity Set (Extension):

e_1 = (John Smith, 55, 80000)
e_2 = (Joe Doe, 40, 20000)
e_3 = (Jane Doe, 27, 30000)
.
.
.

Fig 3.4 Entity type and Entity set

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 is grouped into an entity set, which is also called the **extension** of the entity type.

Key Attributes of an Entity Type

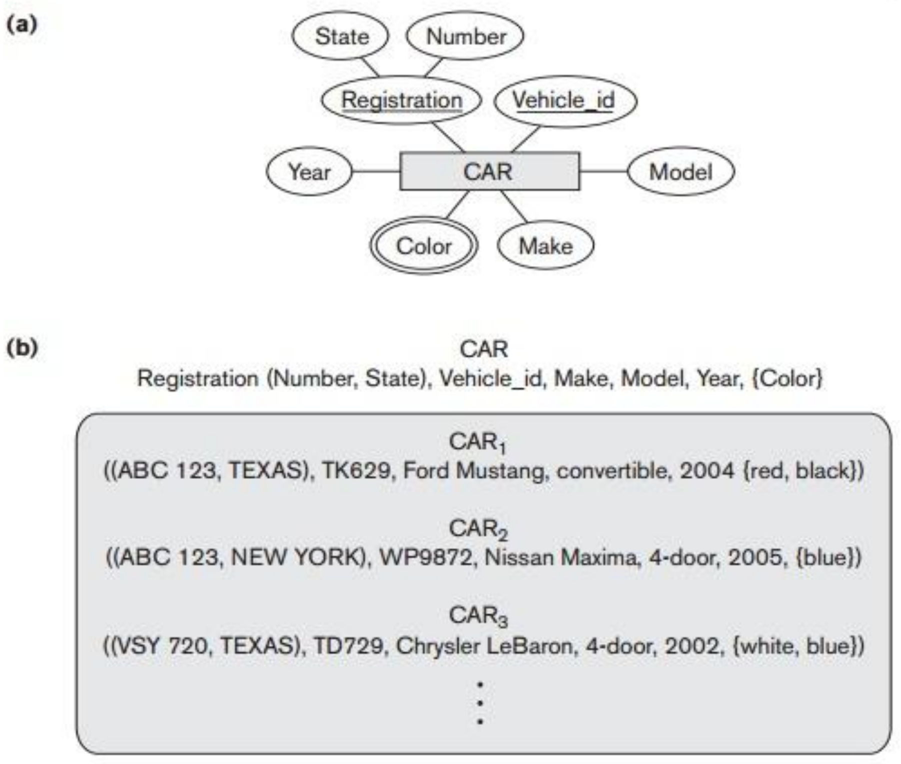
An important constraint on the entities of an entity type is the key or uniqueness constraint on attributes. An entity type usually has one or more attributes whose values are distinct for each individual entity in the entity set. Such an attribute is called a key attribute, and its values can be used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

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

Composite key

Sometimes several attributes together form a key, meaning that the combination of the attribute values must be distinct for each entity. Such a key is called as Composite key.

Figre3.5
The CAR entity type with two key attributes, Registration and Vehicle_id. (a) ER diagram notation. (b) Entity set with three entities.



CAR
Registration (Number, State), Vehicle_id, Make, Model, Year, {Color}

CAR$_1$
((ABC 123, TEXAS), TK629, Ford Mustang, convertible, 2004 {red, black})

CAR$_2$
((ABC 123, NEW YORK), WP9872, Nissan Maxima, 4-door, 2005, {blue})

CAR$_3$
((VSY 720, TEXAS), TD729, Chrysler LeBaron, 4-door, 2002, {white, blue})

⋮

Some entity types have more than one key attribute. For example, each of the Vehicle_id and Registration attributes of the entity type CAR (Figure 3.5) is a key in its own right. The Registration attribute is an example of a composite key formed from two simple component attributes, State and Number, 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: if 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.

3.3.3 Initial Conceptual Design of the COMPANY Database

We can now define the entity types for the COMPANY database, based on the requirements described.

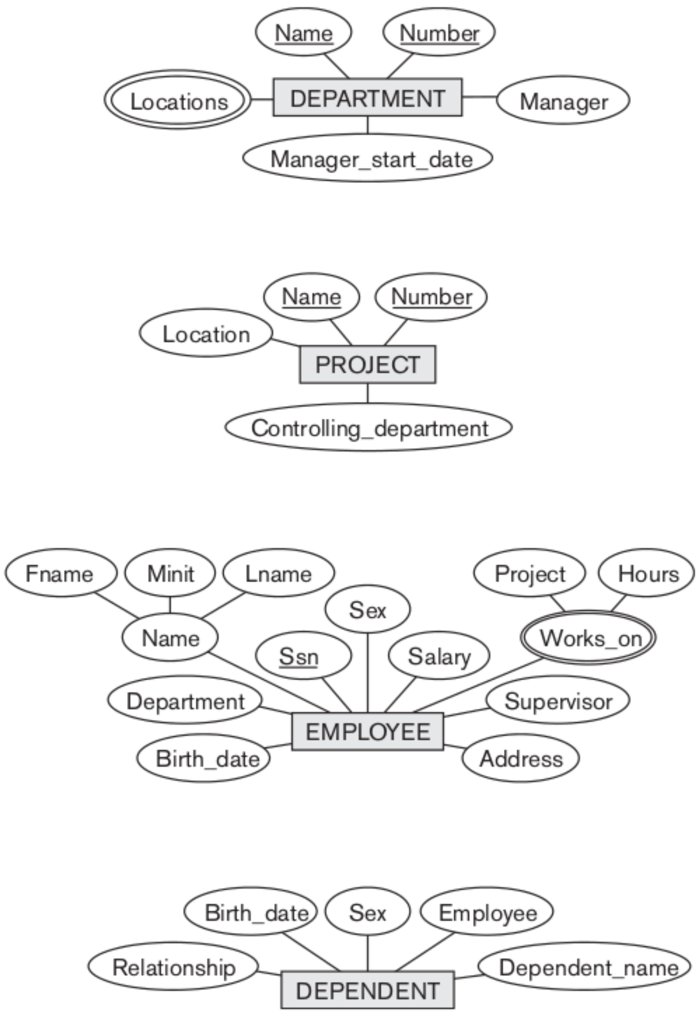


Figure 3.6
Preliminary design of entity types for the COMPANY database. Some of the shown attributes will be refined into relationships.

1. An entity type DEPARTMENT with attributes Name, Number, Locations, Manager, and Manager_start_date. Locations is the only multivalued attribute. We can specify that both Name and Number are (separate) key attributes because each was specified to be unique.

2. An entity type PROJECT with attributes Name, Number, Location, and Controlling_department. Both Name and Number are (separate) key attributes.

3. An entity type EMPLOYEE with attributes Name, Ssn, Sex, Address, Salary, Birth_date, Department, and Supervisor. Both Name and Address may be composite attributes; however, this was not specified in the requirements. We must go back to the users to see if any of them will refer to the individual components of Name—First_name, Middle_initial, Last_name—or of Address. In our example, Name is modeled as a composite attribute, whereas Address is not, presumably after consultation with the users.

4. An entity type DEPENDENT with attributes Employee, Dependent_name, Sex, Birth_date, and Relationship (to the employee).

3.4 Relationship Types, Relationship Sets, Roles, and Structural Constraints

3.4.1 Relationship Types, Sets, and Instances

A relationship type R among n entity types E1, E2, . . . , En defines a set of associations—or a relationship set—among entities from these entity types.

Mathematically, the relationship set R is a set of relationship instances ri, where each ri associates n individual entities (e1, e2, . . . , en), and each entity ej in ri is a member of entity set Ej , 1 ≤ j ≤ n.

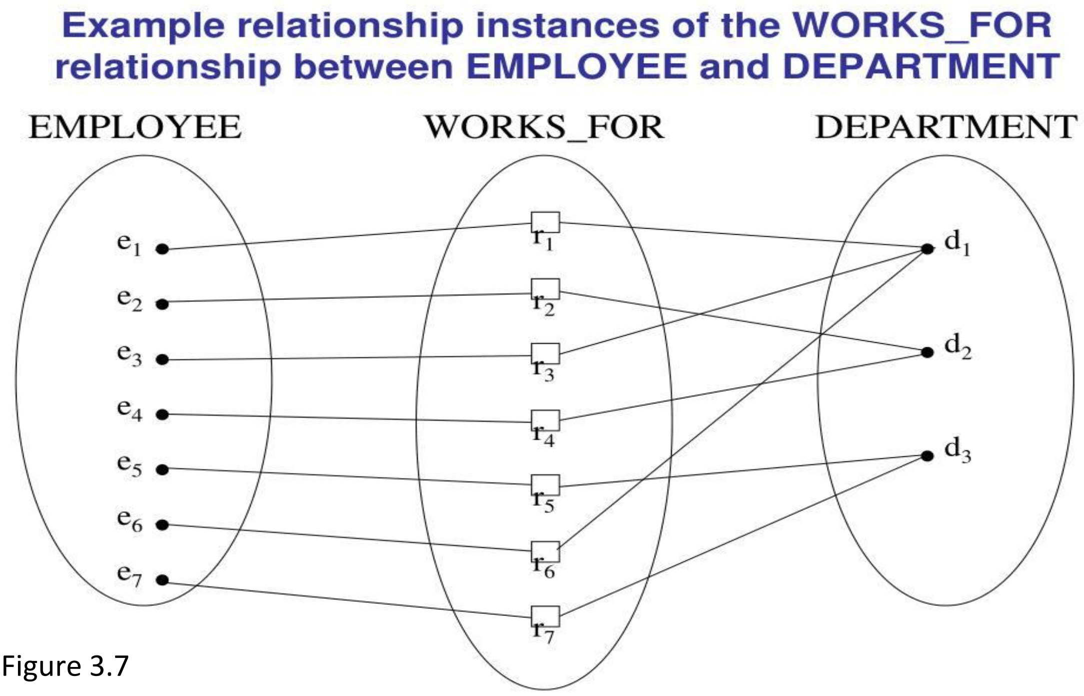


Figure 3.7

For example, consider a relationship type WORKS_FOR between the two entity types EMPLOYEE and DEPARTMENT, which associates each employee with the department for which the employee works. Each relationship instance in the relationship set WORKS_FOR associates one EMPLOYEE entity and one DEPARTMENT entity. Figure 3.7 illustrates this example, where each relationship instance ri is shown connected to the EMPLOYEE and DEPARTMENT entities that participate in ri.

3.4.2 Relationship Degree, Role Names, and Recursive Relationships

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**.

Relationships as Attributes: In case of binary relationship types we can think of representing relationship as an attribute to the corresponding entity type.

Role Names: 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 it helps to explain what the relationship means.

Ex: The WORKS_FOR relationship type, EMPLOYEE plays the role of employee or worker and DEPARTMENT plays the role of department or employer.

Recursive Relationships: 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 the role that each participating entity plays. Such relationship types are called **recursive relationships** or **self-referencing relationships**.

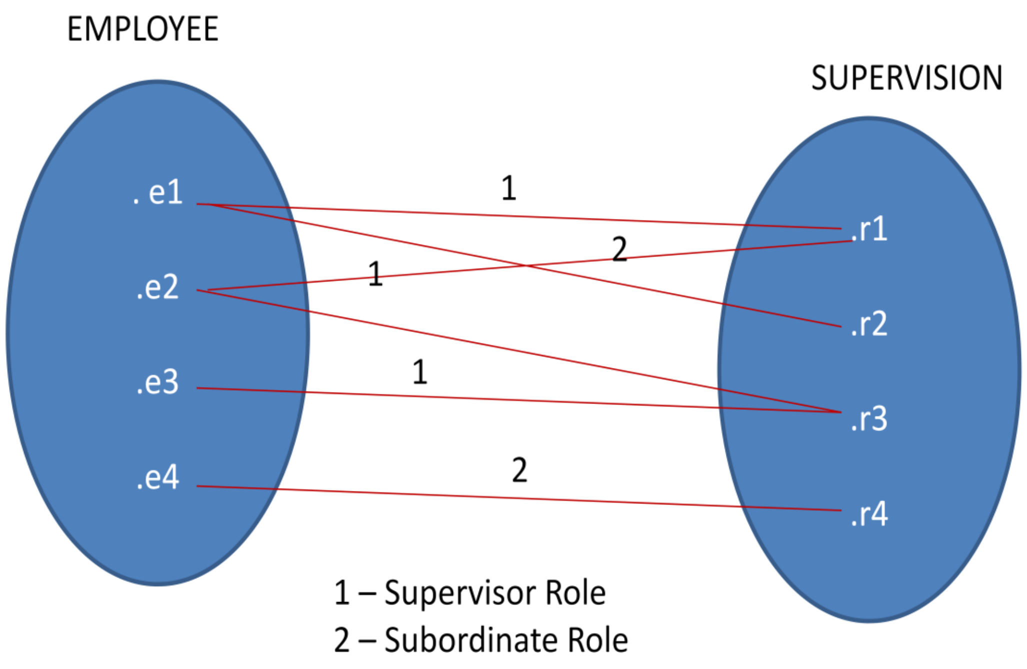


Figure 3.8 Recursive Relationships.

3.4.3 Constraints on Binary Relationship Types

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 miniworld situation that the relationships represent.

We can distinguish two main types of binary relationship constraints:

- Cardinality ratio
- Participation

Cardinality Ratios for Binary Relationships: The cardinality ratio for a binary relationship specifies the maximum number of relationship instances that an entity can participate in.

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

For example, in the WORKS_FOR binary relationship type, DEPARTMENT:EMPLOYEE is of cardinality ratio 1:N, meaning that each department can be related to any number of employees but an employee can be related to at most one department.

An example of a 1:1 binary relationship is MANAGES (Figure 3.9), which relates a department entity to the employee who manages that department. This represents the miniworld constraints that—at any point in time—an employee can manage at most one department and a department can have at most one manager. The relationship type WORKS_ON (Figure 3.10) 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.

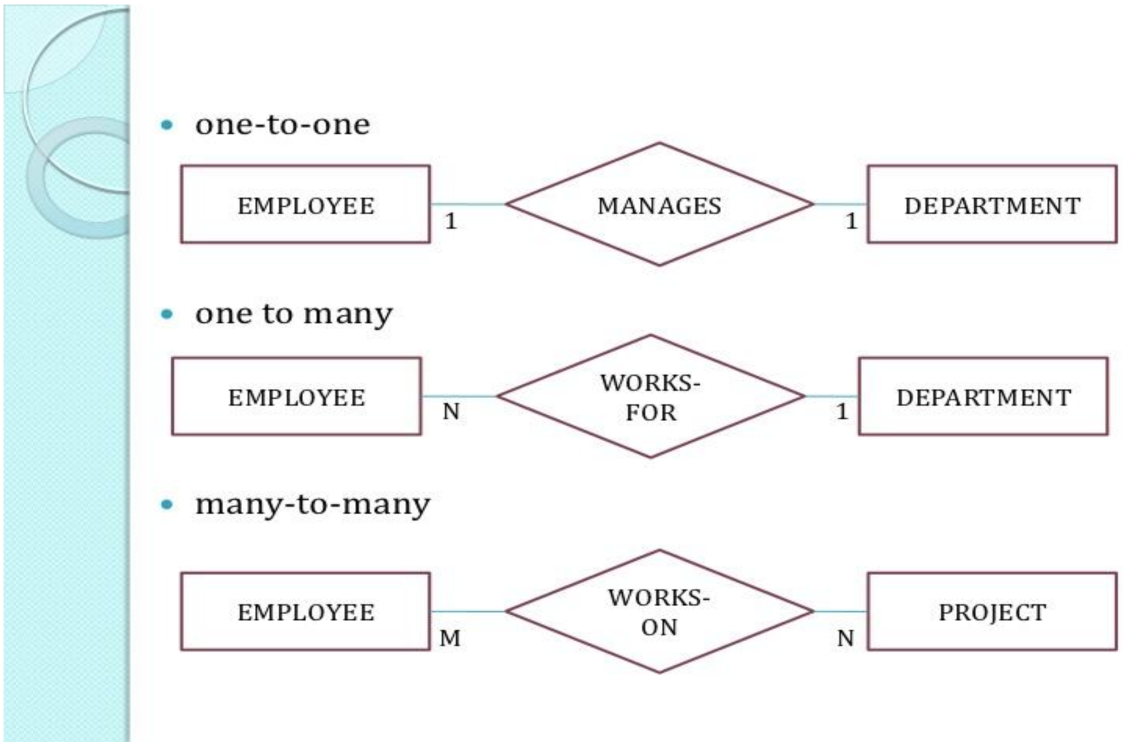


Figure 3.9 : Types of cardinality ratios

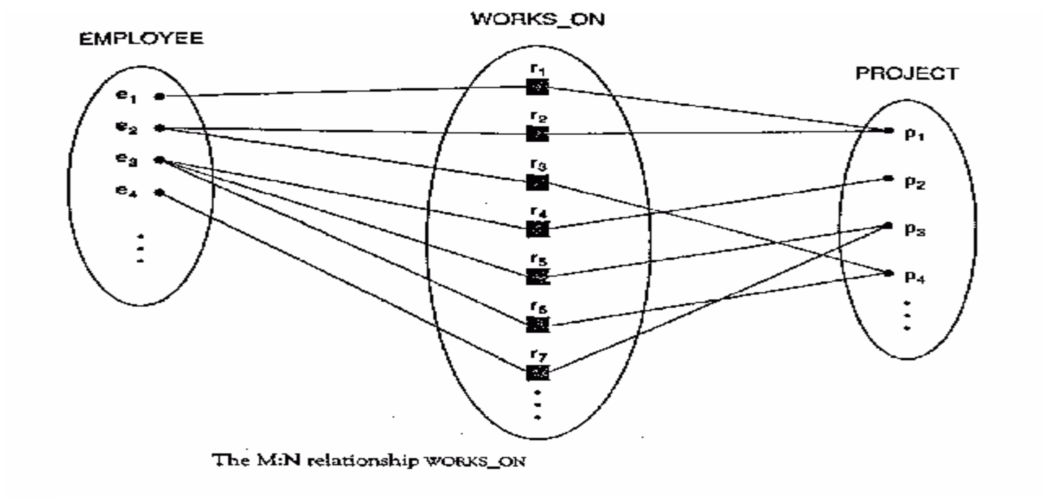


Figure 3.10: M:N cardinality ratio

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. This constraint specifies the minimum number of relationship instances that each entity can participate in and is sometimes called the minimum cardinality constraint.

There are two types of participation constraints—**total** and **partial.**

If a company policy states that every employee must work for a department, then an employee entity can exist only if it participates in at least one WORKS_FOR relationship instance (Figure 3.9). 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**.

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 some department entity via MANAGES, but not necessarily all.

3.4.4 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 a particular employee works on a particular project, we can include an attribute Hours for the WORKS_ON relationship type in Figure 3.10. Another example is to include the date on which a manager started managing a department via an attribute Start_date for the MANAGES relationship type in Figure 3.9.

The attributes of 1:1 or 1:N relationship types can be migrated to one of the participating entity types. For a 1:N relationship type, a relationship attribute can be migrated only to the entity type on the N-side of the relationship. For M:N (many-to-many) 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.

3.5 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—which include all the examples discussed so far—are 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 one 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.

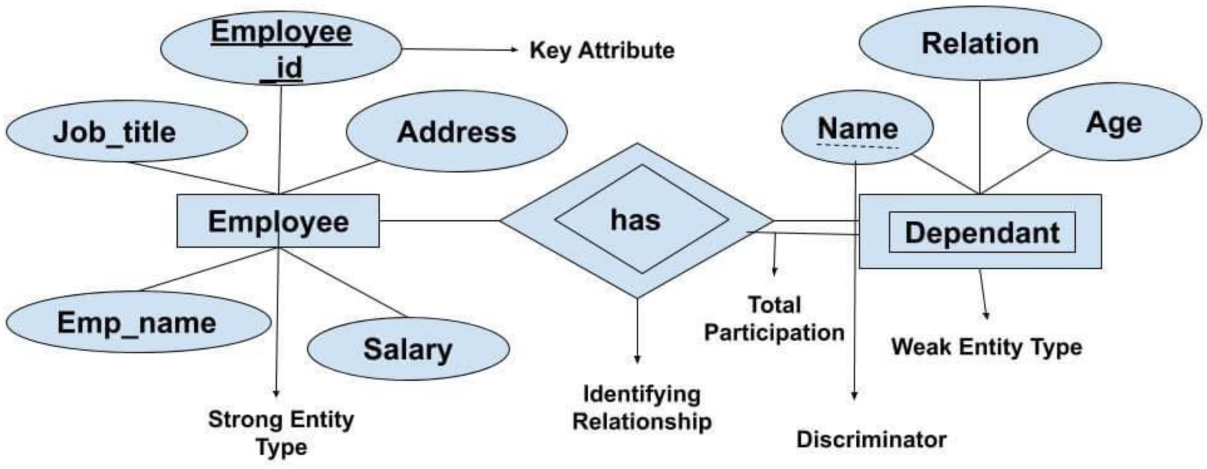


Figure 3.11: Weak entity type

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. In our example, the attributes of DEPENDENT are Name (the first name of the dependent), Birth_date, Sex, and Relationship (to the employee). Two dependents of two distinct employees may, by chance, have the same values for Name, Birth_date, Sex, and Relationship, but they are still distinct entities.

A weak entity type normally has a partial key, which is the attribute that can uniquely identify weak entities that are related to the same owner entity. 2 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.

3.6Refining the ER Design for the COMPANY Database

We can now refine the database design by changing the attributes that represent relationships into relationship types. The cardinality ratio and participation constraint of each relationship type are determined from the requirements.

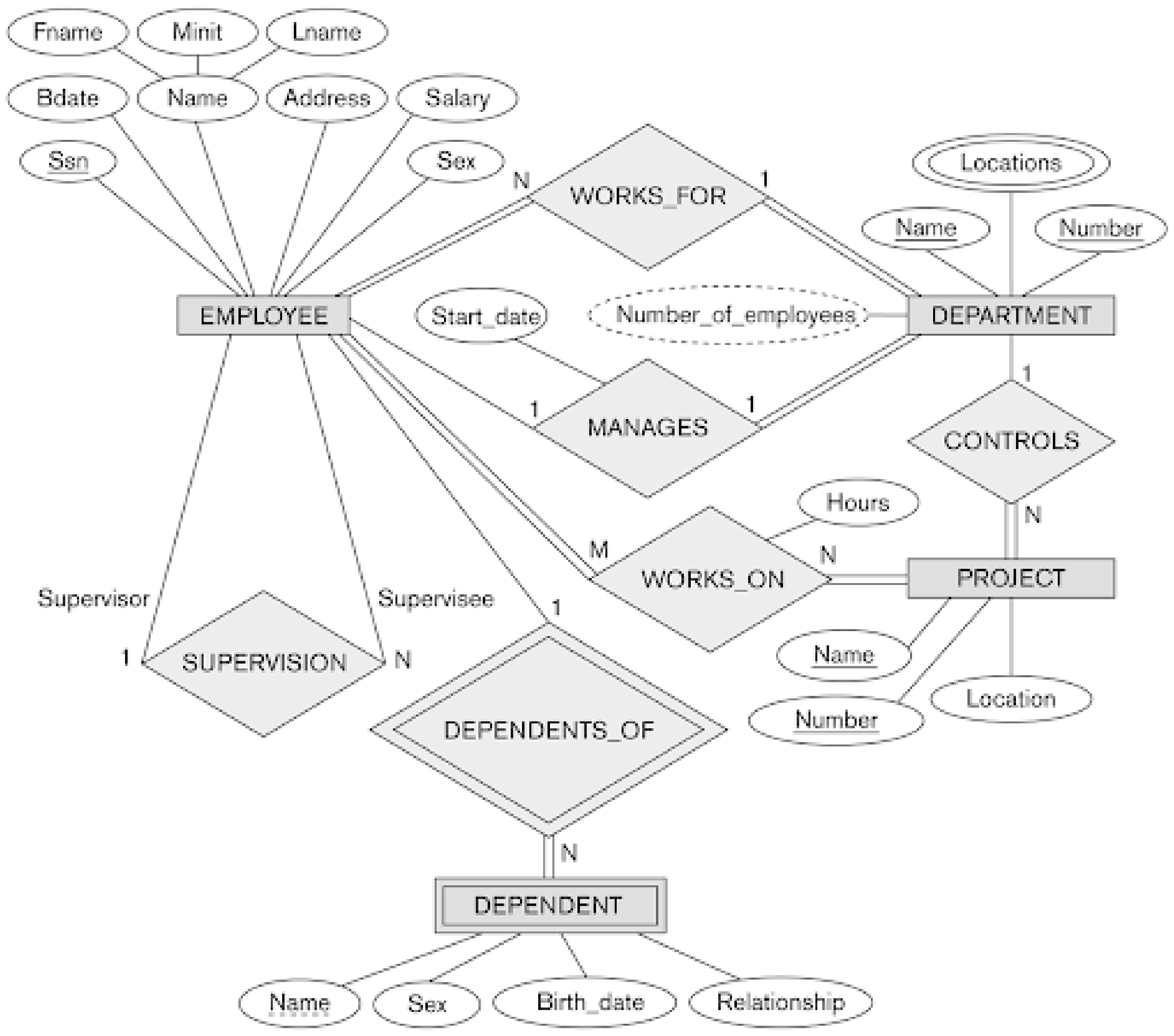


Figure 3.12

An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter.

In our example, we specify the following relationship types:

■ MANAGES, which is a 1:1(one-to-one) 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.13 The attribute Start_date is assigned to this relationship type.

■ WORKS_FOR, a 1:N (one-to-many) relationship type between DEPARTMENT and EMPLOYEE. Both participations are total.

■ 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 indicates that some departments may control no projects.

■ 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.

■ WORKS_ON, determined to be an M:N (many-to-many) 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.

■ 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.

3.7ER Diagrams, Naming Conventions, and Design Issues

3.7.1 Summary of Notation for ER Diagrams

In ER diagrams the emphasis is more on representing schemas rather than instances. Schemas changes rarely.

- Entity types are shown in rectangular boxes.
  EMPLOYEE, DEPARTMENT.
- Relationship types are shown in diamond-shaped boxes attached to the participating entity types with straightlines.
  WORKS_FOR, MANAGES.
- Attributes are shown in ovals. Each attribute is attached by a straight line to its entity type.
- Multi-valued attributes are shown in double ovals.
- Key attributes have their names underlined.
- Derived attributes are shown in dotted oval.
- Weak entity types are represented by double line rectangle.
- Partial key of weak entity type is underlined with a dotted line.
- Cardinality ratio of each relationship is specified by attaching 1,M,N on participating edge.
- Participation constraints-
- Total participation - double lines
- Partial participation – single lines.

3.7.2 Proper naming of Schema constructs

- Entity types should be Singular names.
- Entity type and relationship type are to be written in Uppercase letters.
- Attributes have their initial letter capitalized.
- Rolenames are in lowercase letters.
- Relationship uses verbs as their names and Entity type uses nouns.

3.7.3 Design Choices for ER Conceptual Design

In general, the schema design process should be considered an iterative refinement process, where an initial design is created and then iteratively refined until the most suitable design is reached. Some of the refinements that are often used include the following:

- A concept may be first modeled as an attribute and then refined into a relationship because it is determined that the attribute is a reference to another entity type.
- An attribute that exists in several entity types may be elevated or promoted to an independent entity type. For example, suppose that each of several entity types in a UNIVERSITY database, such as STUDENT, INSTRUCTOR, and COURSE, has an attribute Department in the initial design; the designer may then choose to create an entity type DEPARTMENT with a single attribute Dept_name and relate it to the three entity types (STUDENT, INSTRUCTOR, and COURSE) via appropriate relationships. Other attributes/relationships of DEPARTMENT may be discovered later.
- An inverse refinement to the previous case may be applied—for example, if an entity type DEPARTMENT exists in the initial design with a single attribute Dept_name and is related to only one other entity type, STUDENT. In this case, DEPARTMENT may be reduced or demoted to an attribute of STUDENT.

3.8 Specialization and Generalization

Specialization and generalization are fundamental concepts in database modelling that are useful for establishing superclass-subclass relationships.

3.8.1 Specialization

Specialization is the process of defining a set of subclasses of an entity type; this entity type is called the **superclass** of the specialization. The set of subclasses that forms a specialization is defined on the basis of some distinguishing characteristic of the entities in the superclass.

Specialization is a top-down approach in which a higher-level entity is divided into multiple *specialized* lower-level entities. In addition to sharing the attributes of the higher-level entity, these lower-level entities have *specific* attributes of their own. Specialization is usually used to find subsets of an entity that has a few different or additional attributes.

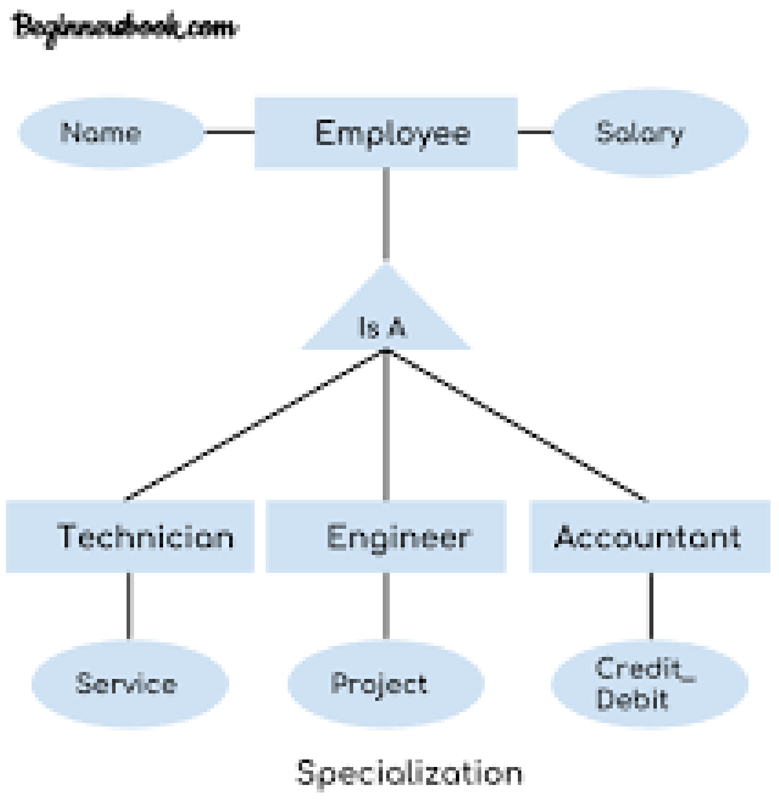


Figure 3.13 Specialization

For example, the set of subclasses {ACCOUNTANT, ENGINEER, TECHNICIAN} is a specialization of the superclass EMPLOYEE that distinguishes among employee entities based on the job type of each employee. We may have several specializations of the same entity type based on different distinguishing characteristics.

3.8.2 Generalization

A reverse process of abstraction in which we suppress the differences among several entity types, identify their common features, and generalize them into a single superclass of which the original entity types are special subclasses.

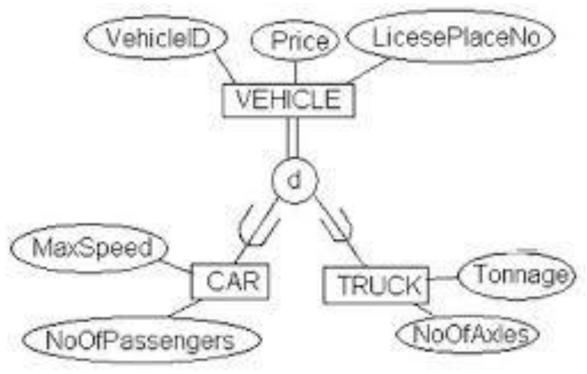


Figure 3.14 Generalization

Consider the entity types CAR and TRUCK. Because they have several common attributes, they can be generalized into the entity type VEHICLE, as shown in figure 3.14 . Both CAR and TRUCK are now subclasses of the generalized superclass VEHICLE. We use the term **generalization** to refer to the process of defining a generalized entity type from the given entity types.

The generalization process can be viewed as being functionally the inverse of the specialization process; we can view {CAR, TRUCK} as a specialization of VEHICLE rather than viewing VEHICLE as a generalization of CAR and TRUCK.