MODULE I

Introduction: Data: structured, semi-structured and unstructured data, Concept & Overview of DBMS, Data Models, Database Languages, Database Administrator, Database Users, Three Schema architecture of DBMS. Database architectures and classification.

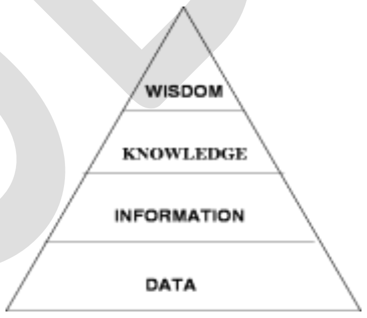
Entity-Relationship Model: Basic concepts, Design Issues, Mapping Constraints, Keys, Entity-Relationship Diagram, Weak Entity Sets, Relationships of degree greater than 2.

Reference:

Elmasri R. and S. Navathe, Database Systems: Models, Languages, Design and Application Programming, Pearson Education, 2013.

Data, Information and Knowledge

- **Data** represents unorganized and unprocessed facts.
    - Known facts that can be recorded and have implicit meaning
    - Usually data is static in nature.
    - It can represent a set of discrete facts about events.
    - Data is a prerequisite to information.
- **Information**
    - Information can be considered as an aggregation of data (processed data) which makes decision making easier.
    - Information has usually got some meaning and purpose.
- **Knowledge**
    - By knowledge we mean *human understanding of a subject matter that has been acquired through proper study and experience*.
    - Knowledge is usually based on learning, thinking, and proper understanding of the problem area.
    - Knowledge is not information and information is not data.
    - Knowledge is derived from information in the same way information is derived from data.



Structured, Semi-structured and Unstructured Data

Structured data is information, usually text files, displayed in titled columns and rows which can easily be ordered and processed by data mining tools. It has been reformatted and its elements organized into a data structure so that elements can be addressed, organized and accessed in various combinations to make better use of the information. This could be visualized as a perfectly organized filing cabinet where everything is identified, labeled and easy to access.

Unstructured data has not been organized into a format that makes it easier to access and process. Unstructured data is raw and unorganized.

Eg.

Emails

Word Processing Files

PDF files

Spreadsheets

Digital Images

Video

Audio

Social Media Posts

Semi-structured data lies somewhere between the two. It is not organized in a complex manner that makes sophisticated access and analysis possible; however, it may have information associated with it that allows elements contained to be addressed.

Database and DBMS

A **database** is a collection of related data. The computer program used to manage and query a database is known as a **database management system (DBMS).** So a database is a collection of related data that we can use for

- Defining - specifying *types* of data
- Constructing - storing & populating
- Manipulating - querying, updating, reporting

Features of a Database

- It is a persistent (stored) collection of related data.
- The data is input (stored) only once.
- The data is organised (in some fashion).
- The data is accessible and can be queried (effectively and efficiently).

Drawbacks of Using File Systems

- ***Data redundancy and inconsistency***

Due to availability of multiple file formats, storage in files may cause duplication of information in different files.

- ***Difficulty in accessing data***

In order to retrieve, access and use stored data, need to write a new program to carry out each new task.

- ***Data isolation***

To isolate data we need to store them in multiple files and different formats.

- ***Integrity problems***

Integrity constraints (E.g. account balance > 0) become part of program code which has to be written every time. It is hard to add new constraints or to change existing ones.

- ***Atomicity of updates***

Failures of files may leave database in an inconsistent state with partial updates carried out.
E.g. transfer of funds from one account to another should either complete or not happen at all.

- ***Concurrent access by multiple users***

Concurrent access of files is needed for better performance and it also true that uncontrolled concurrent accesses of files can lead to inconsistencies.
E.g. two people reading a balance and updating it at the same time.

Database Management System (DBMS)

A **database management system (DBMS)** is a collection of programs 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.

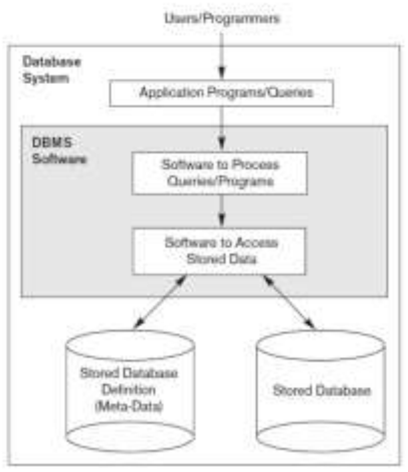
- **Defining** a database involves specifying the data types, structures, and constraints of the data to be stored in the database. The database definition or descriptive information is also stored by the DBMS in the form of a database catalog or dictionary; it is called **meta-data**.
- **Constructing**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

An **application program** accesses the database by sending queries or requests for data to the DBMS.

A **query** typically causes some data to be retrieved.

A **transaction** may cause some data to be read and some data to be written into the database.

Simplified Database Environment


Users/Programmers
Database System
Application Programs/Queries
DBMS Software
Software to Process Queries/Programs
Software to Access Stored Data
Stored Database Definition (Meta-Data)
Stored Database

Characteristics of the Database Approach

■ **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. The DBMS software must work equally well with any number of database applications—for example, a university database, a banking database, or a company database—as long as the database definition is stored in the catalog.

■ **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. User application programs can operate on the data by invoking these operations through their names and arguments, regardless of how the operations are implemented. This may be termed **program-operation independence**.

The characteristic that allows program-data independence and program-operation independence is called **data abstraction**. A DBMS provides users with a conceptual representation of data that does not include many of the details of how the data is stored or how the operations are implemented.

■ **Support of multiple views of the data**

A database typically has many users, each of whom may require a different perspective or view of the database. A view may be a subset of the database or it may contain virtual data that is derived from the database files but is not explicitly stored. Some users may not need to be aware of whether the data they refer to is stored or derived. A multiuser DBMS whose users have a variety of distinct applications must provide facilities for defining multiple views.

■ **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 agents try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one agent at a time for assignment to a passenger. These types of applications are generally called online transaction processing (OLTP) applications. A fundamental role of multiuser DBMS software is to ensure that concurrent transactions operate correctly and efficiently.

Database Users

Actors on Scene

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.

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.

3. End Users

- **Casual end users** occasionally access the database, but they may need different information each time.

- **Naive** or **parametric end users :** constantly querying and updating the database, using standard types of queries and updates called **canned transactions**—that have been carefully programmed and tested
- **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

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 standard canned transactions that meet these requirements. **Application programmers** implement these specifications as programs; then they test, debug, document, and maintain these canned transactions.

Behind the Scene

1. **DBMS system designers and implementers:** design and implement the DBMS modules and interfaces as a software package
2. **Tool developers** design and implement **tools**—the software packages that facilitate database modelling and design, database system design, and improved performance.
3. **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.

Advantages of using a DBMS

- **Controlling Redundancy**

  Data redundancy means duplication of data. The redundancy in storing the same data multiple times leads to several problems. The storage space is wasted and files that represent the same data may become inconsistent. 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.

- **Controlling Inconsistency**

  An inconsistent database provides incorrect or conflicting information. By controlling redundancy the inconsistency is also controlled.

- **Data integrity and security**

  If data is always accessed through the DBMS, the DBMS can enforce integrity constraints on the data. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, the DBMS can enforce access controls that govern what data is visible to different classes of users. Data security refers to protection of data.

- **Facilitate sharing of data**

  Individual pieces of data in the database may be shared among several different users in the sense that each of those users may have access to the same piece of data and each of them may use it for different purpose. When several users share the data, centralizing the administration of data can offer significant improvements.

- **Concurrent access and crash recovery**

  A DBMS schedules concurrent accesses to the data in such a manner that users can think of the data as being accessed by only one user at a time. Further, the DBMS protects users from the effects of system failures.

- **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.

- **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. The DBA can enforce standards in a centralized database environment more easily than in an environment where each user group has control of its own data files and software.
- **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.

Data Abstraction, Data Model and Structure of a Database

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.

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.
Structure of a database: The data types, relationships, and constraints that apply to the data.
Most data models also include a set of **basic operations** for specifying retrievals and updates on the database.

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

- Provide concepts that are close to the way many users perceive data
- 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, for example, a works-on relationship between an employee and a project.

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.
- Concepts provided by low-level data models are generally meant for computer specialists, not for end users.
- Describe how data is stored as files in the computer by representing information such as record formats, record orderings, and access paths.
- An **access path** is a structure that makes the search for particular database records efficient.
- An **index** is an example of an access path that allows direct access to data using an index term or a keyword.

Representational (or Implementation) Data Models

- Between high level and low level data models.
- 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.
- Hide many details of data storage on disk but can be implemented on a computer system directly.
- Used most frequently in traditional commercial DBMSs.

- Represent data by using record structures and hence are sometimes called **record-based data models**.

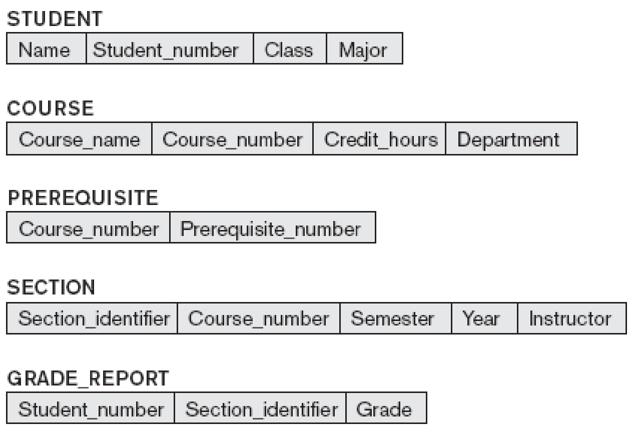
Object Data Model

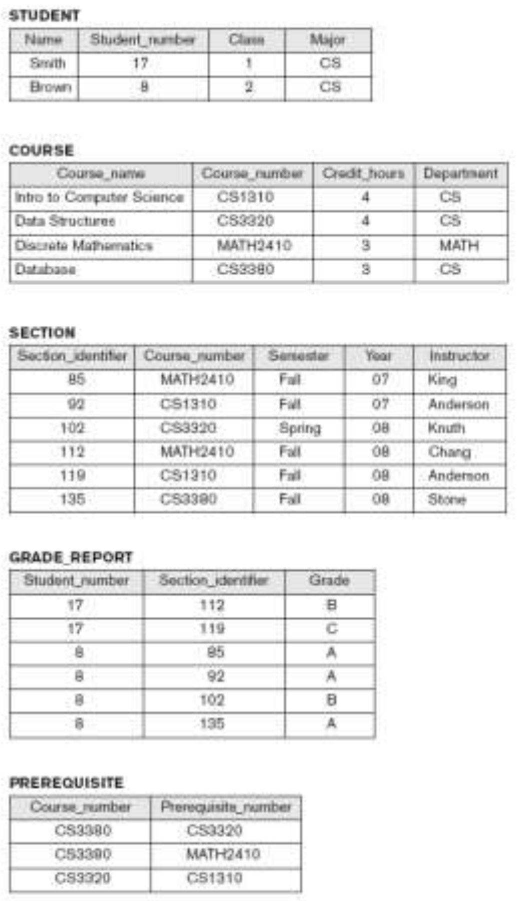
- An example of a new family of higher-level implementation data models that are closer to conceptual data models.
- A standard for object databases called the ODMG object model has been proposed by the Object Data Management Group (ODMG).

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


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



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
A database that stores student and course information

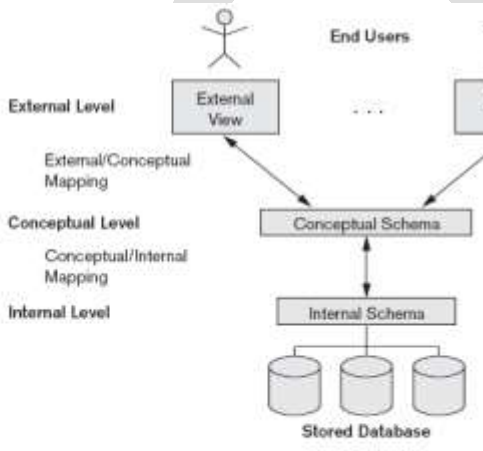
Database State

The actual data in a database may change quite frequently. For example, the database shown in the Figure changes every time we add a new student or enters a new 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.

The distinction between database schema and database state is very important. When we **define** a new database, we specify its database schema only to the DBMS. At this point, the corresponding database state is the *empty state* with no data. We get the *initial state* of the database when the database is first **populated** or **loaded** with the initial data. From then on, every time an update operation is applied to the database, we get another database state. At any point in time, the database has a *current state*.

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 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 is called an **extension** of the schema. The schema is not supposed to change frequently, it is not uncommon that changes occasionally need to be applied to the schema as the application requirements change. This is known as **schema evolution**.

Three Schema Architecture



Three of the four important characteristics of the database approach, (1) use of a catalog to store the database description (schema) 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 are achieved by the **three-schema architecture**

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:

1. The **internal level**

 - has an **internal schema**, which describes the physical storage structure of the database.

- uses a physical data model

- 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.

- hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints.

- Usually, a representational data model is used to describe the conceptual schema when a database system is implemented.

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.

- typically implemented using a representational data model.

The three schemas are only *descriptions* of data; the stored data that *actually* exists is at the physical level only. In a DBMS based on 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**.

Data Independence

The three-schema architecture can be used to further 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:

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), to change constraints, or to reduce the database (by removing a record type or data item).

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. Changes to the internal schema may be needed because some physical files were reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update.

Database Languages and Interfaces

Once the design of a database is completed and a DBMS is chosen to implement the database, the first step is to specify conceptual and internal schemas for the database and any mappings between the two.

- **Data Definition Language**

In many DBMSs where no strict separation of levels is maintained, one language, called the **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.

- **Storage Definition Language**

In DBMSs where a clear separation is maintained between the conceptual and internal levels, the DDL is used to specify the conceptual schema only. Another language, the **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**

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

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 set of operations or a language called the **data manipulation language** (**DML**) for these purposes.

There are two main types of DMLs.

High-level or nonprocedural DML - can be used on its own to specify complex database operations concisely. Many DBMSs allow high-level DML statements either to be entered interactively from a display monitor or terminal or to be embedded in a general-purpose programming language.

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; therefore, they are called **set-at-a-time** or **set-oriented** DMLs.

A query in a high-level DML often specifies *which* data to retrieve rather than *how* to retrieve it; therefore, such languages are also called **declarative**.

Whenever DML commands, whether high level or low level, are embedded in a general-purpose programming language, that language is called the **host language** and the DML is called the **data sublanguage**. On the other hand, a high-level DML used in a standalone interactive manner is called a **query language**.

DBMS Interfaces

User-friendly interfaces provided by a DBMS may include the following:

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. The query is composed step-by step by picking options from a menu that is displayed by the system. Pull-down menus are a very popular technique in **Web-based user interfaces**. They are also 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 can fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Many DBMSs have **forms specification languages**, which are special languages that help programmers specify such forms.

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. In many cases, GUIs utilize both menus and forms. Most GUIs use a **pointing device**, such as a mouse, to select 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. The natural language interface refers to the words in its schema, as well as to the set of standard words in its dictionary, to interpret the request.

Speech Input and Output. Limited use of speech as an input query and speech as an answer to a question or result of a request is becoming commonplace. 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.

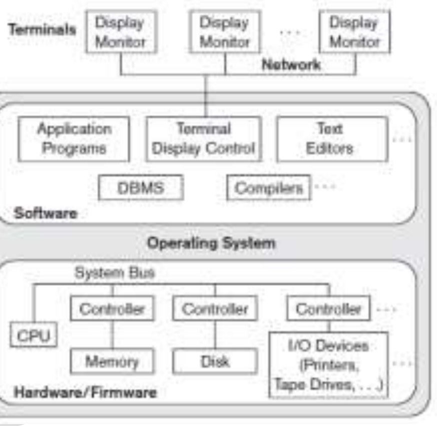
Interfaces for Parametric Users. Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. For example, a teller is able to use single function keys to invoke routine and repetitive transactions such as account deposits or withdrawals, or balance inquiries.

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.

Database Architectures

Centralized DBMSs Architecture

DBMS itself was still a centralized DBMS in which all the DBMS functionality, application program execution, and user interface processing were carried out on one machine.



Basic Client/Server Architectures

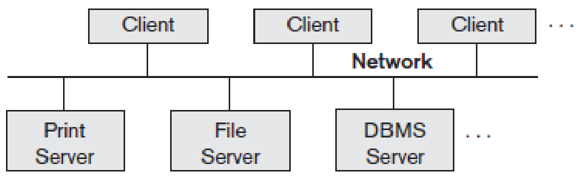
The client/server architecture was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, data base servers, Web servers, e-mail servers, and other software and equipment are connected via a network. A client in this framework is typically a user machine that provides user interface capabilities and local processing. When a client requires access to additional functionality— such as database access— that does not exist at that machine, it connects to a server that provides the needed functionality. A server is a system containing both hardware and software that can provide services to the client machines, such as file access, printing, archiving, or database access.

Two-Tier Client/Server Architectures for DBMSs

In relational database management systems (RDBMSs), many of which started as centralized systems, the system components that were first moved to the client side were the user interface and application programs. Because SQL provided a standard language for RDBMSs, this created a logical dividing point between client and server. Hence, the query and transaction functionality related to SQL processing remained on the server side. In such an architecture, the server is often called a **query server** or **transaction server** because it provides these two functionalities. In an RDBMS, the server is also often called an **SQL server**.

The user interface programs and application programs can run on the client side. When DBMS access is required, the program establishes a connection to the DBMS (which is on the server side); once the connection is created, the client program can communicate with the DBMS. A standard called **Open Database Connectivity** (**ODBC**) provides an **application programming interface** (**API**), which allows client-side programs to call the DBMS, as long as both client and server machines have the necessary software installed. Most DBMS vendors provide ODBC drivers for their systems. A client program can actually connect to several RDBMSs and send query and transaction requests using the ODBC API, which are then processed at the server sites. Any query results are sent back to the client program, which can process and display the results as needed. A related standard for the Java programming language, called **JDBC**, has also been defined. This allows Java client programs to access one or more DBMSs through a standard interface.

The architecture described here is called **two-tier architecture** because the software components are distributed over two systems: client and server. The advantages of this architecture are its simplicity and seamless compatibility with existing systems. The emergence of the Web changed the roles of clients and servers, leading to the three-tier architecture.



Three-Tier and n-Tier Architectures for Web Applications

Many Web applications use an architecture called the **three-tier architecture**, which adds an intermediate layer between the client and the database server. This intermediate layer or **middle tier** is called the **application server** or the **Web server**, depending on the application. This server plays an intermediary role by running application programs and storing business rules (procedures or constraints) that are used to access data from the database server. It can also improve database security by checking a client's credentials before forwarding a request to the database server. Clients contain GUI interfaces and some additional application-specific business rules. The intermediate server accepts requests from the client, processes the request and sends database queries and commands to the database server, and then acts as a conduit for passing (partially) processed data from the database server to the clients, where it may be processed further and filtered to be presented to users in GUI format. Thus, the *user interface*, *application rules*, and *data access* act as the three tiers.

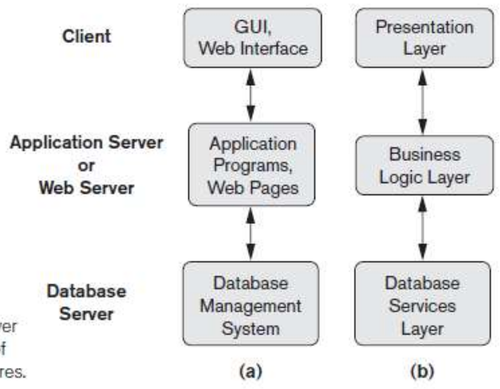


Figure 2.7
Logical three-tier client/server architecture, with a couple of commonly used nomenclatures.

Basic Concepts of ER Diagrams

Entity Types, Entity Sets, Attributes, and Keys

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

Entities and Attributes

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 (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).

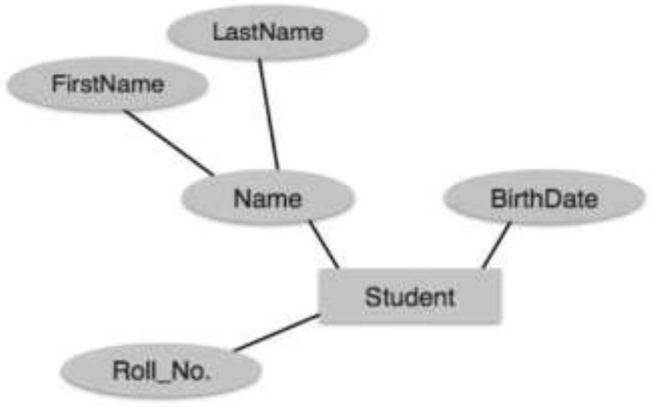
Student
Teacher
Projects

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.

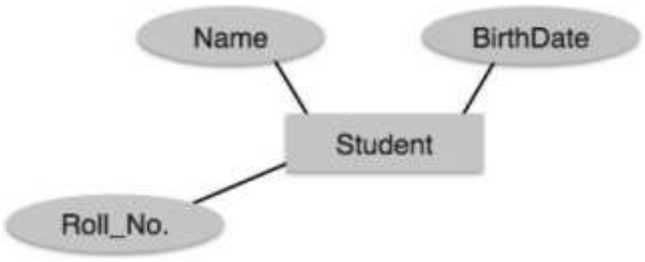
Types of Attributes

Composite versus Simple (Atomic) Attributes.

Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meanings. For example, the Address attribute of the EMPLOYEE entity can be subdivided into Street_address, City, State, and Zip.



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



Single-Valued versus Multivalued Attributes.

Most attributes have a single value for a particular entity; such attributes are called **single-valued**. For example, Age is a single-valued attribute of a person.

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. Different people can have different *numbers* of *values* for an attribute. Such attributes are called **multivalued**. 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 three values, if we assume that a car can have three colors at most.

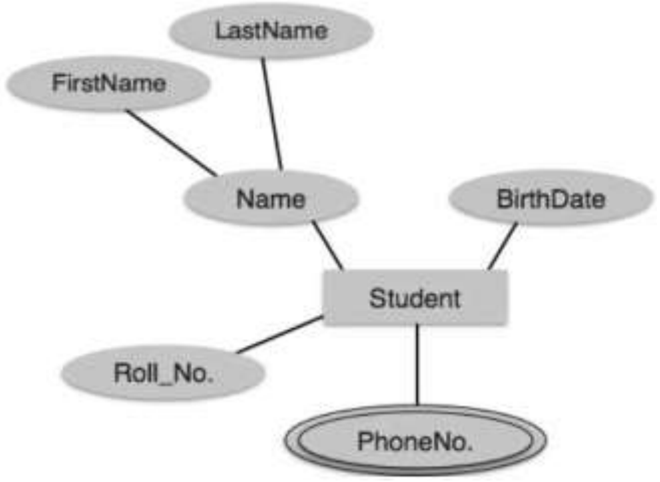


Fig: Multivalued Attributes

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**.

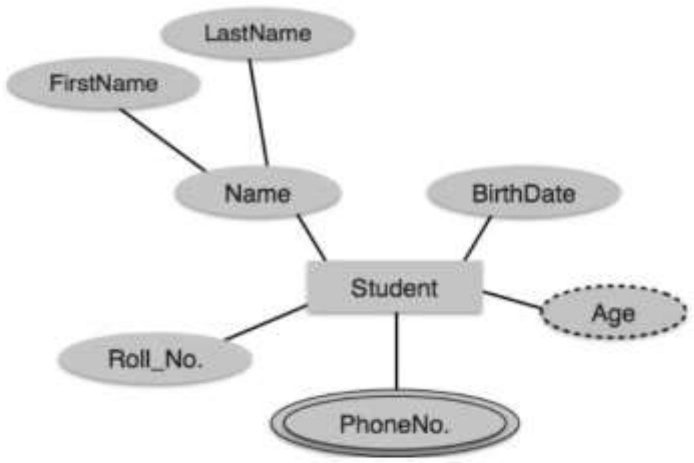


Fig: Derived attributes

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. An address of a single-family home would have NULL for its Apartment_number attribute, and a person with no college degree would have NULL for College_degrees.

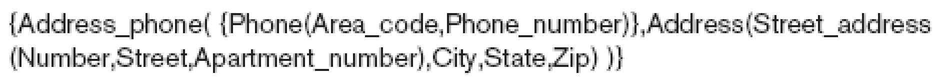
NULL can also be used if we do not know the value of an attribute for a particular entity—for example, if we do not know the home phone number of 'John Smith'.

The meaning of the former type of NULL is *not applicable*, whereas the meaning of the latter is *unknown*. 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 instance, 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 Home_phone attribute of a person is NULL.

Complex Attributes.

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**.

For example, if a person can have more than one residence and each residence can have a single address and multiple phones, an attribute Address_phone for a person can be specified as shown in Figure.


{Address_phone({Phone(Area_code,Phone_number)},Address(Street_address
(Number,Street,Apartment_number),City,State,Zip))}

Entity Types and Entity Sets.

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. 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.

Key Attributes of an Entity Type.

Key is an attribute or collection of attributes that uniquely identifies an entity among entity set. For example, the roll_number of a student makes him/her identifiable among students.

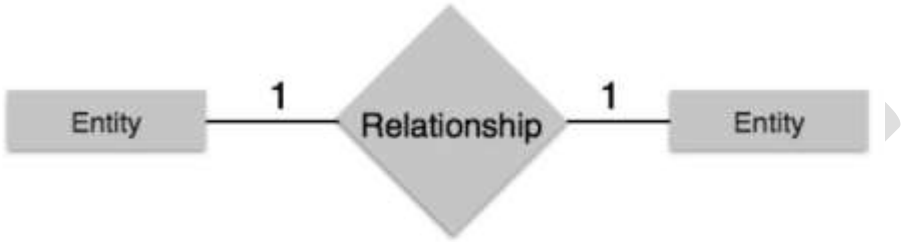
- **Super Key:** A set of attributes (one or more) that collectively identifies an entity in an entity set.
- **Candidate Key:** A minimal super key is called a candidate key. An entity set may have more than one candidate key.
- **Primary Key:** A primary key is one of the candidate keys chosen by the database designer to uniquely identify the entity set.

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.

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. As for the case of entity types and entity sets, a relationship type and its corresponding relationship set are customarily referred to by the same name, R.



Degree of a Relationship Type.

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

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 participating 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 the role that each participating entity plays. Such relationship types are called **recursive relationships**.

The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity set. 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*).

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* and *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.

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 (that is, employs) any number of employees, but an employee can be related to (work for) only one 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—at any point in time—an employee can manage one department only and a department can have one manager only.

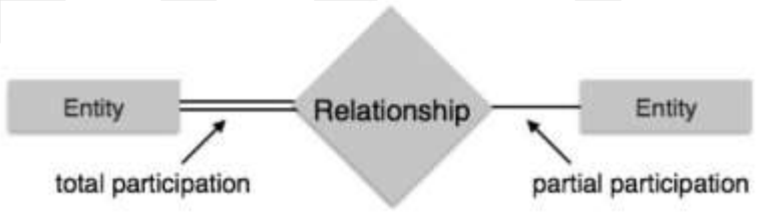
The relationship type WORKS_ON is of cardinality ratio M:N, because the mini-world rule is that an employee can work on several projects and a project can have several employees.

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. 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.



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

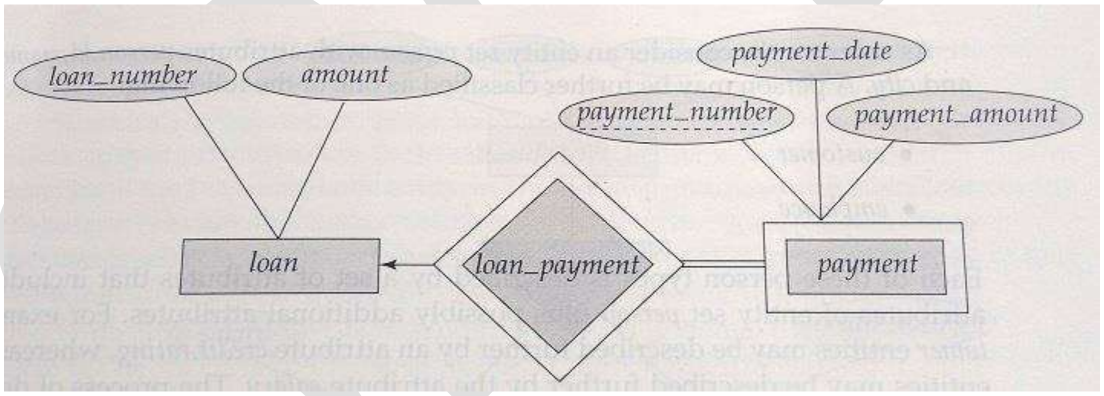
Weak Entity Types

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 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.

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. 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 attribute 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.



An Example

The COMPANY database keeps track of a company's employees, departments, and projects.

Miniworld Description

■ 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.

■ We store each employee's name, Social Security number, 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. We keep track of the current number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee (who is another employee).

■ 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.

Initial Conceptual Design of the COMPANY Database

We can now define the entity types for the COMPANY database, based on the requirements. According to the requirements listed, we can identify four entity types—one corresponding to each of the four items in the specification.

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

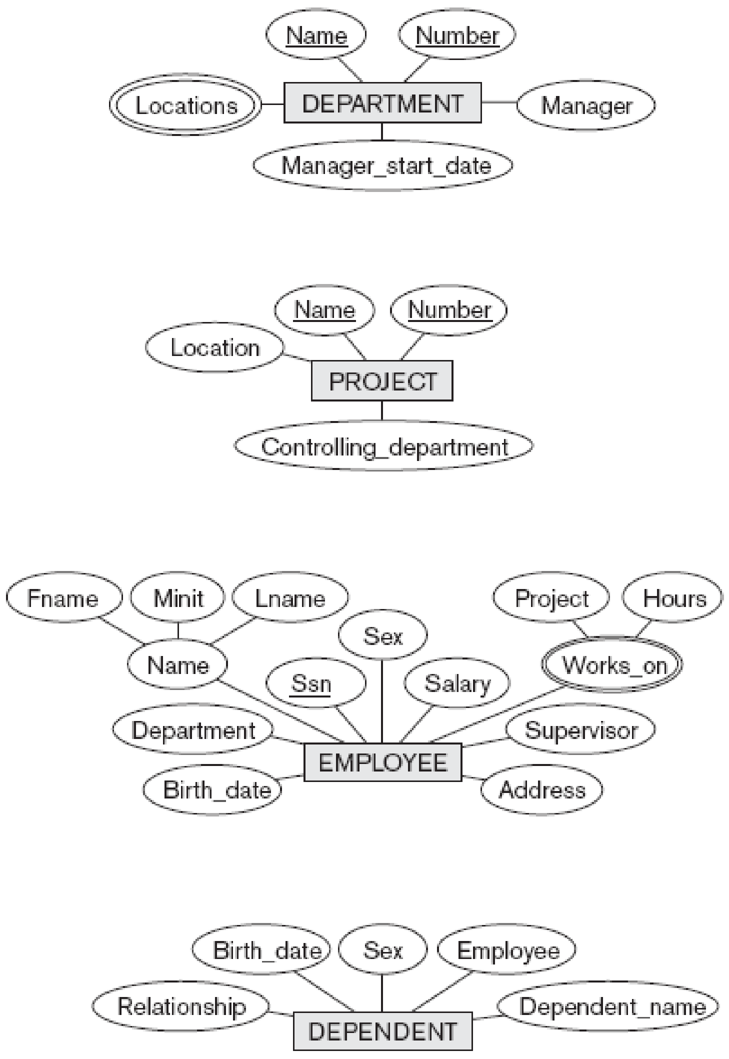


Figure 7.8
Preliminary design of entity types for the COMPANY database. Some of the shown attributes will be refined into relationships.

Refining 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. If some cardinality ratio or dependency cannot be determined from the requirements, the users must be questioned further to determine these structural constraints.

In our example, we specify the following relationship types:

■ 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 Start_date is assigned to this relationship type.

■ WORKS_FOR, a 1:N 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 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.

After specifying the above six relationship types, we remove from the entity all attributes that have been refined into relationships. These include Manager and Manager_start_date from DEPARTMENT; Controlling_department from PROJECT; Department, Supervisor, and Works_on from EMPLOYEE; and Employee from DEPENDENT.

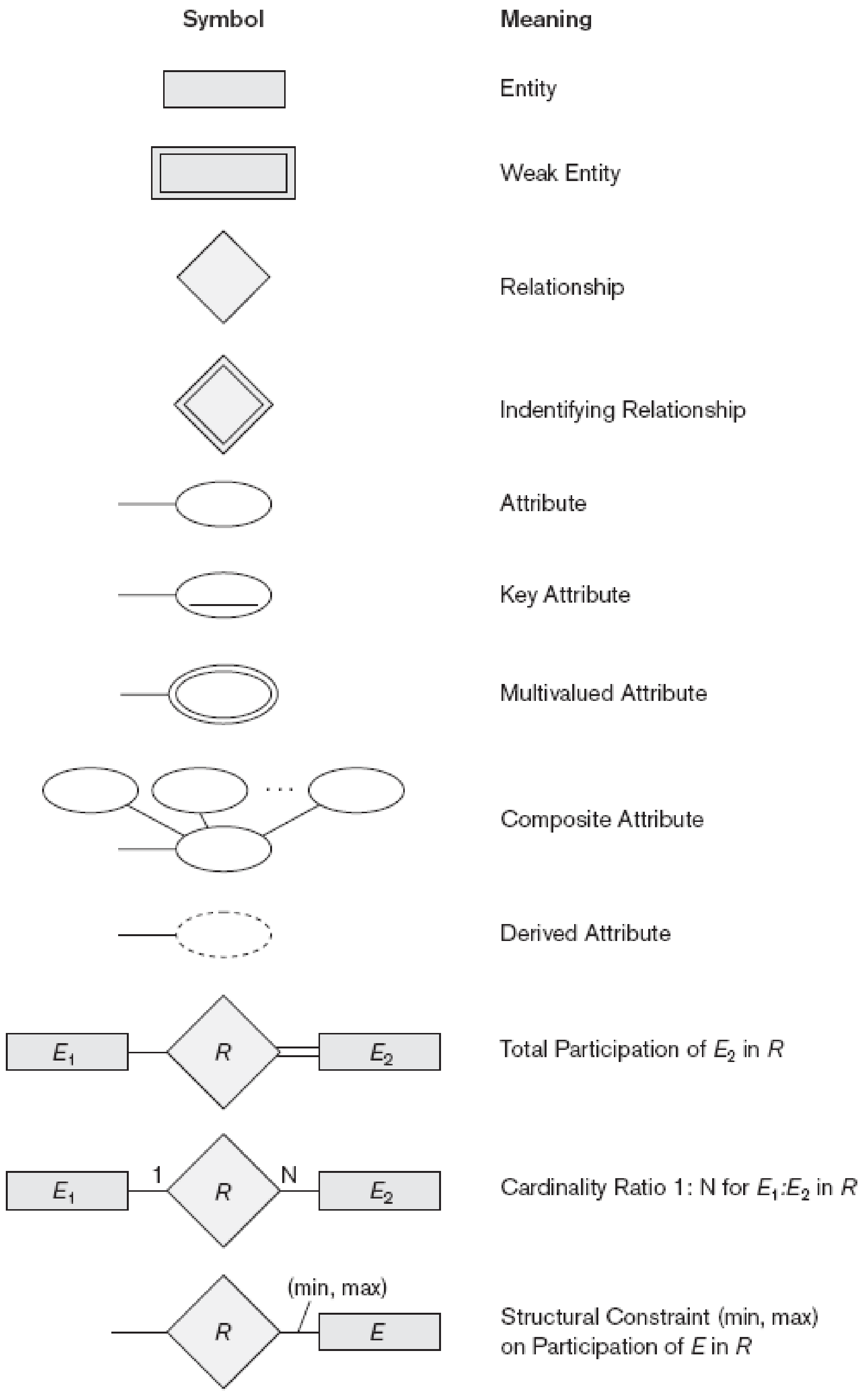
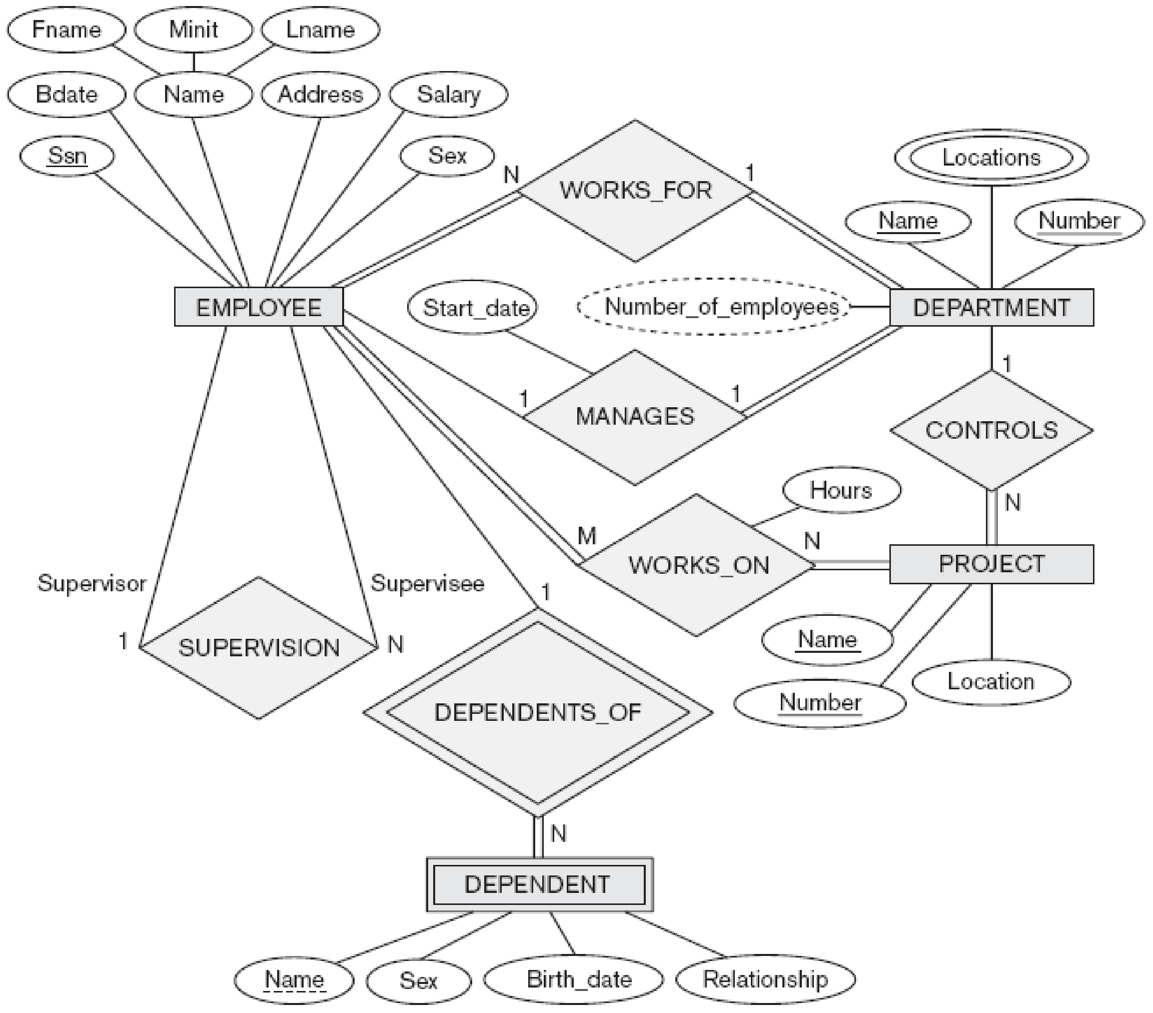
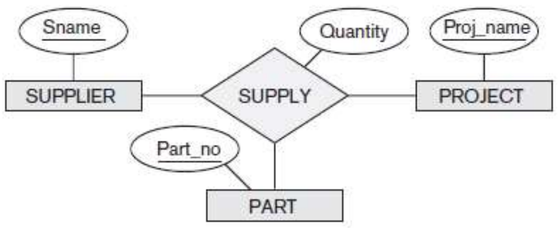


Figure 7.14
Summary of the notation for ER diagrams.



Relationship Types of Degree Higher than Two

The **degree** of a relationship type as the number of participating entity types and relationship type of degree two is called *binary* and a relationship type of degree three is called *ternary*.



The relationship set of SUPPLY is a set of relationship instances (*s*, *j*, *p*), where *s* is a SUPPLIER who is currently supplying a PART *p* to a PROJECT *j*. In general, a relationship type *R* of degree *n* will have *n* edges in an ER diagram, one connecting *R* to each participating entity type.

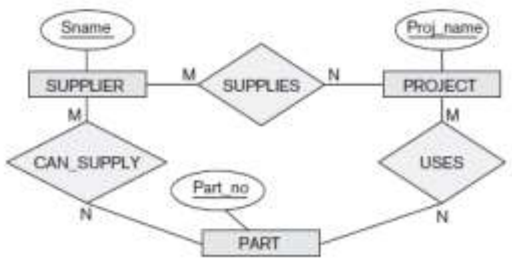


Figure shows an ER diagram for three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. In general, a ternary relationship type represents different information than do three binary relationship types.

Consider the three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. Suppose that CAN_SUPPLY, between SUPPLIER and PART, includes an instance (*s*, *p*) whenever supplier *s can supply* part *p* (to any project); USES, between PROJECT and PART, includes an instance (*j*, *p*) whenever project *j* uses part *p*; and SUPPLIES, between SUPPLIER and PROJECT, includes an instance (*s*, *j*) whenever supplier *s* supplies *some part* to project *j*. The existence of three relationship instances (*s*, *p*), (*j*, *p*), and (*s*, *j*) in CAN_SUPPLY, USES, and SUPPLIES, respectively, does not necessarily imply that an instance (*s*, *j*, *p*) exists in the ternary relationship SUPPLY, because the *meaning is different*.

It is often tricky to decide whether a particular relationship should be represented as a relationship type of degree *n* or should be broken down into several relationship types of smaller degrees. The designer must base this decision on the semantics or meaning of the particular situation being represented. The typical solution is to include the ternary relationship *plus* one or more of the binary relationships, if they represent different meanings and if all are needed by the application.

try it now

A KTU STUDENTS PLATFORM

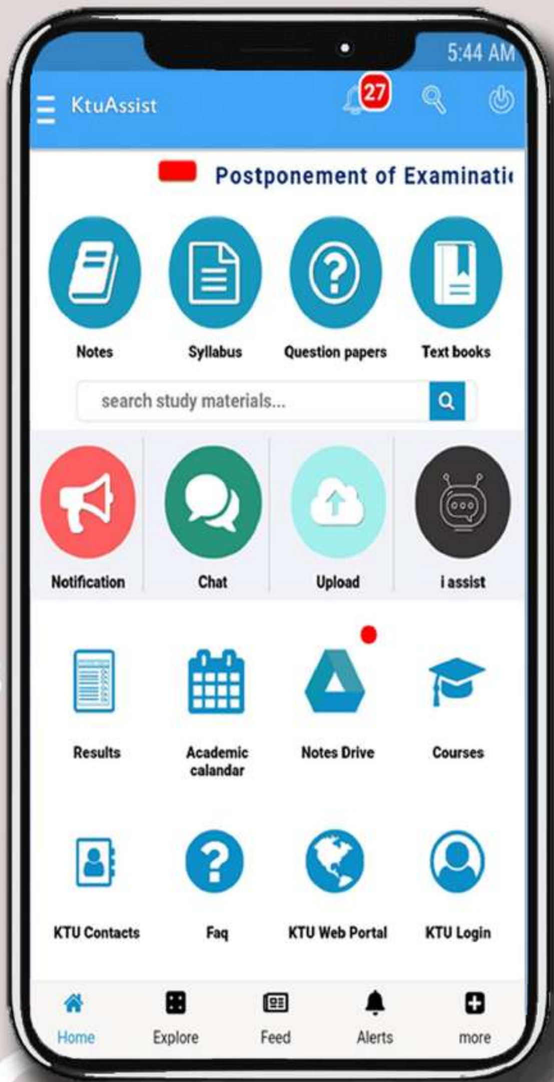
SYLLABUS

NOTES

TEXT BOOKS

QUESTION PAPERS

KTU NOTIFICATION



DOWNLOAD IT FROM GOOGLE PLAY

CHAT

CALENDAR

FAQ

LOGIN

MUCH MORE

DOWNLOAD APP

ktuassist.in

instagram.com/ktu_assist

facebook.com/ktuassist