UNIT 1

The word *database* is so commonly used that we must begin by defining what a database is.

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

Data means known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. You may have recorded this data in an indexed address book or you may have stored it on a hard drive, using a personal computer and software such as Microsoft Access or Excel.
This collection of related data with an implicit meaning is a database.

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.

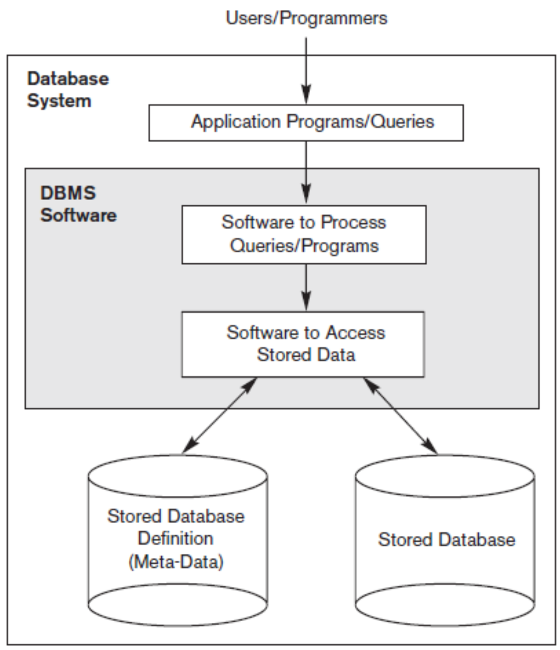
The database definition or descriptive information is also stored by the DBMS in the form of a database catalog or dictionary it is called **meta-data**.
Constructing the database is the process of storing the data on some storage medium that is controlled by the DBMS.
Manipulating a database includes 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.

<u>Characteristics of the Database Approach</u>
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

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.

A general-purpose DBMS software package is not written for a specific database application. Therefore, it must refer to the catalog to know the structure of the files in a specific database, such as the type and format of data it will access.

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



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**.

Support of Multiple Views of the Data

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.

Data Abstraction:

A **data model** is used to hide storage details and present the users with a conceptual view of the database. Programs refer to the data model constructs rather than data storage details

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. 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.*

*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.*

Database Users

Users may be divided into-

Those who actually use and control the database content, and those who design, develop and maintain database applications (called "*Actors on the Scene*"), and those who design and develop the DBMS software and related tools, and the computer systems operators (called "*Workers Behind the Scene*").

Actors on the scene

Database Designers:

- Responsible to define the content, the structure, the constraints, and functions or transactions against the database. They must communicate with the end-users and understand their needs.

Database administrators:

- Responsible for authorizing access to the database, for coordinating and monitoring its use, acquiring software and hardware resources, controlling its use and monitoring efficiency of operations.

End-users: They use the data for queries, reports and some of them update the database content. End-users can be categorized into:

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

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

- **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 tax package that stores a variety of personal financial data for tax purposes.

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.

Workers behind the Scene

These persons are typically not interested in the database content itself., call them the *workers behind the scene*

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.** 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 modeling and design, database system design, and improved performance. Tools are optional packages that are often purchased separately.

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 the DBMS Approach

1. *Controlling Redundancy* in data
2. *Sharing of data among multiple users.*
3. *Restricting unauthorized access to data.*
4. *Providing Persistent Storage for Program Objects*
5. *Providing Storage Structures for Efficient Query Processing.*
6. *Providing Backup and Recovery Services.*
7. *Providing Multiple Interfaces*
8. *Representing Complex Relationships among data.*
9. *Enforcing Integrity Constraints on the Database.*
10. *Drawing Inferences and Actions using rules*

***Controlling Redundancy* in data storage** 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: 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. 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.

Restricting unauthorized access to data. 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 only authorized persons are allowed to access the data. In addition, some users may only be permitted to retrieve data, whereas others are allowed to retrieve and update. A DBMS should provide a **security and authorization subsystem.**

Providing Persistent Storage for Program Objects

Databases can be used to provide **persistent storage** for program objects and data structures. 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.

The persistent storage of program objects and data structures is an important function of database systems. Traditional database systems often suffered from the so called **impedance mismatch problem**

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 used for this purpose.

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.

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. forms-style interfaces and menu-driven interfaces are used and commonly known as **graphical user interfaces (GUIs)**. Many specialized languages and environments exist for specifying GUIs.

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.

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.

Permitting Inferencing and Actions Using Rules

Some database systems provide capabilities for defining *deduction rules* for *inferencing* new information from the stored database facts

Additional Implications of Using the Database Approach

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.

Reduced Application Development Time. A prime selling feature of the database approach is that developing a new application—such as the retrieval of certain data from the database for printing a new report—takes very little time .

Flexibility. It may be necessary to change the structure of a database as requirements change.

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.

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.

When not to use a DBMS?

there are a few situations in which a DBMS may involve unnecessary overhead costs that would not be incurred in traditional file processing..

High initial investment in hardware, software, and training The generality that a DBMS provides for defining and processing data.

■ Overhead for providing security, concurrency control, recovery, and integrity functions Therefore, it may be more desirable to use regular files under the following circumstances:

- ■ Simple, well-defined database applications that are not expected to change a all.
- ■ Stringent, real-time requirements for some application programs that maynot be met because of DBMS overhead.
- Embedded systems with limited storage capacity, where a general-purpose DBMS would not fit No multiple-user access to data

DATA MODEL

A **data model**—a collection of concepts that can be used to describe the structure of a database

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** provide concepts that are close to the way many users perceive data,

Low-level or **physical data models** provide concepts that describe the details of how data is stored on the computer storage.

these two extremes is a class of **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.

The description of a database is called the **database schema**

STUDENT

Name	Student_number	Class	Major

COURSE

Course_name	Course_number	Credit_hours	Department

Metadata: The DBMS stores the descriptions of the schema constructs and constraints—also called the **meta-data.**

The schema is sometimes called the **intension**, and a database state is called an **extension** of the schema.

Schema Diagram: An ***illustrative*** display of (most aspects of) a database schema.

Schema Construct: A ***component*** of the schema or an object within the schema, e.g., STUDENT, COURSE.

Database State: The actual data stored in a database at a ***particular moment in time***. This includes the collection of all the data in the database or database instance (or occurrence or snapshot).

Database State: Refers to the ***content*** of a database at a moment in time.

Initial Database State: Refers to the database state when it is initially loaded into the system.

Valid State: A state that satisfies the structure and constraints of the database.

Three-Schema Architecture and Data Independence

The goal of the three-schema architecture, illustrated in Figure is to separate the user applications from the physical database.

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.

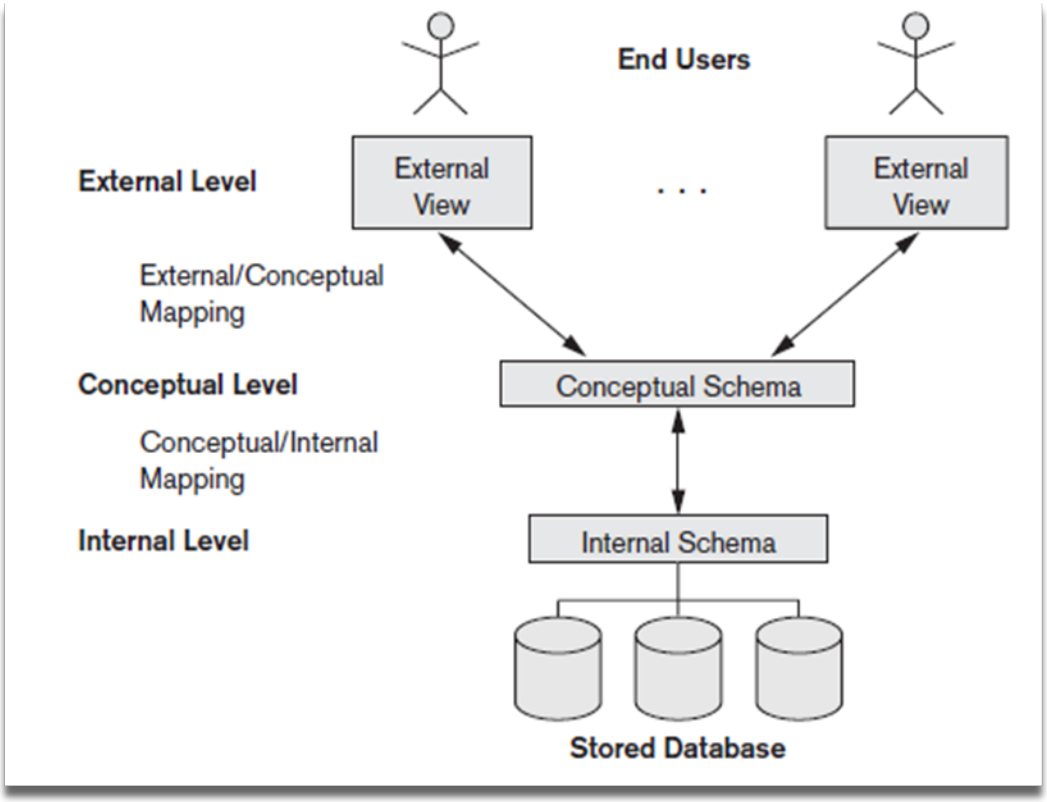


Fig:3 schema architecture

Data Independence

There are two types of data independence:

1. **Logical data independence**
2. **Physical data independence**

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), to change constraints, or to reduce the database (by removing a record type or data item).
2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual 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. If the same data as before remains in the database, we should not have to change the conceptual schema.

Database Languages and Interfaces

The DBMS must provide appropriate languages and interfaces for each Category of users.

DBMS Languages

- Data Definition Language (DDL):
- Storage Definition Language (SDL)
- View Definition Language (VDL)
- Data Manipulation Language (DML)

Data definition language (DDL): *Used by the DBA and database designers* to specify the conceptual schema of a database. 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 (SDL), is used to specify the internal schema. The mappings between the two schemas may be specified in either one of these languages.

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

Data Manipulation Language (DML), Used to specify database retrievals and updates.DML commands (data sublanguage) can be *embedded* in a general-purpose programming language (host language), such as COBOL, C, C++, or Java.

DBMS Interfaces

Stand-alone query language interfaces

- Entering SQL queries at the DBMS interactive SQL interface (e.g. SQL*Plus in ORACLE)

Programmer interfaces for embedding DML in programming languages

User-friendly interfaces

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

- Menu-based, popular for browsing on the web
- Forms-based, designed for navie users
- Graphics-based
  - (Point and Click, Drag and Drop, etc.)
- Natural language: requests in written English
- Combinations of the above

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. Menus do away with the need to memorize the specific commands and syntax of a query language. Pull-down menus are a very popular technique in **Web-based user interfaces**.

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.

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. 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.

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.

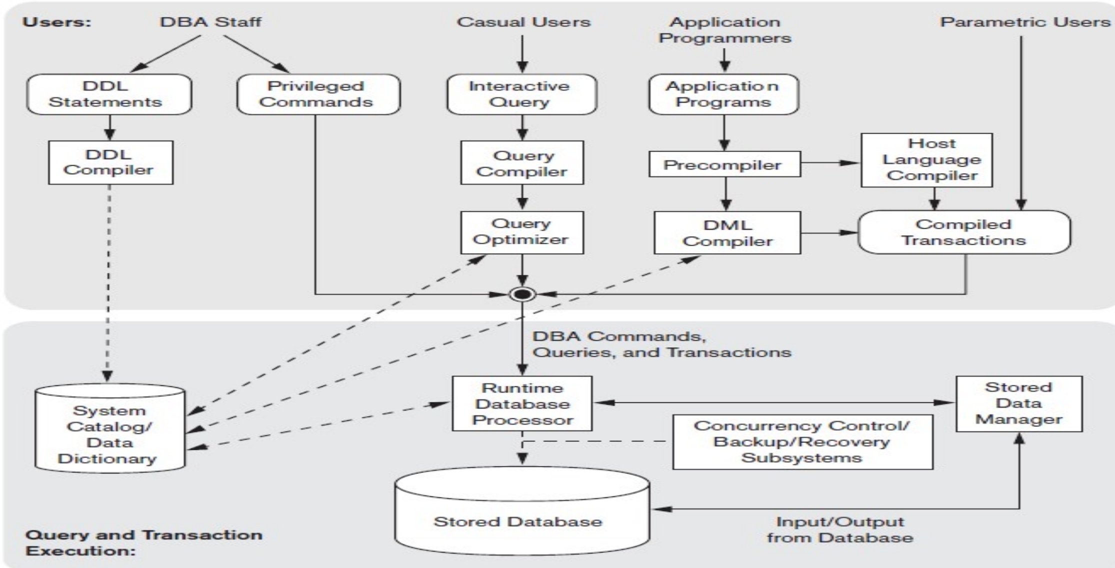
Interfaces for the DBA. Most database systems contain privileged commands that can be used only by the DBA staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structures of a database.

The Database System Environment/typical components of DBMS Module and interactions

A DBMS is a complex software system. the types of software components that constitute a DBMS and the types of computer system software with which the DBMS interacts.

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 internals 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).**

Many DBMSs have their own **buffer management** module to schedule disk



read/write, because this has a considerable effect on performance. top part of Figure 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 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. the **query optimizer** is concerned with the rearrangement and possible reordering of operations, elimination of redundancies, and use of correct algorithms and indexes during execution.
The **precompiler** extracts DML commands from an application program written in a host programming language. We have shown **concurrency control** and **backup and recovery systems** separately as a module in this figure.

The DBMS interacts with the operating system when disk accesses—to the database or to the catalog—are needed. If the computer system is shared by many users, the OS will schedule DBMS disk access requests and DBMS processing along with other processes. On the other hand, if the computer system is mainly dedicated to running the database server, the DBMS will control main memory buffering of disk pages.

<u>Centralized and Client/Server Architectures for DBMSs</u>
Centralized DBMSs Architecture: combines everything into single system including- DBMS software, hardware, application programs and user interface processing software.
The 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. Below Figure illustrates the physical components in a centralized architecture. Gradually, DBMS systems started to exploit the available processing power at the user side, which led to client/server DBMS architectures.

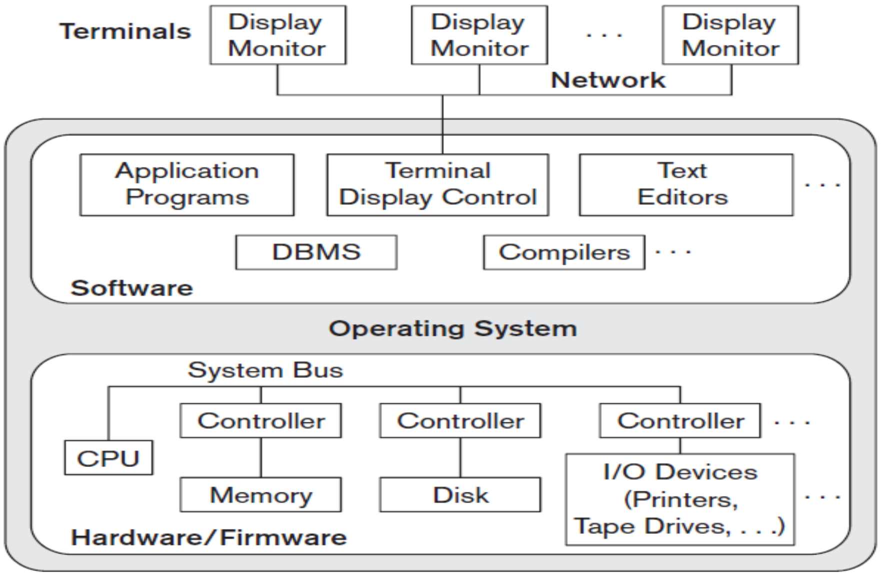


Fig: A physical centralized architecture.

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.

The idea is to define **specialized servers** with specific functionalities. For example, it is possible to connect a number of PCs or small workstations as clients to a **file server** that maintains the files of the client machines. Another machine can be designated as a **printer server** by being connected to various printers; all print requests by the clients are forwarded to this machine.

Web servers or **e-mail servers** also fall into the specialized server category. The resources provided by specialized servers can be accessed by many client machines.

The **client machines** provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications.

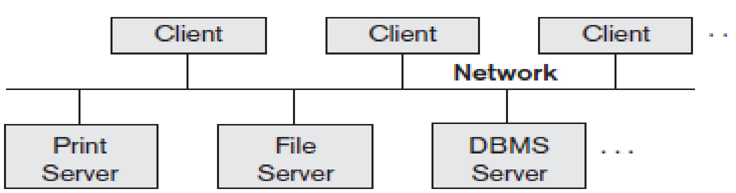


Fig: Logical two-tier client server architecture

The concept of client/server architecture assumes an underlying framework that consists of many PCs and workstations as well as a smaller number of mainframe machines, connected via LANs and other types of computer networks. 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. In general, some machines install only client software, others only server software, and still others may include both client and server software, as illustrated in Figure below. it is more common that client and server software usually run on separate machines.

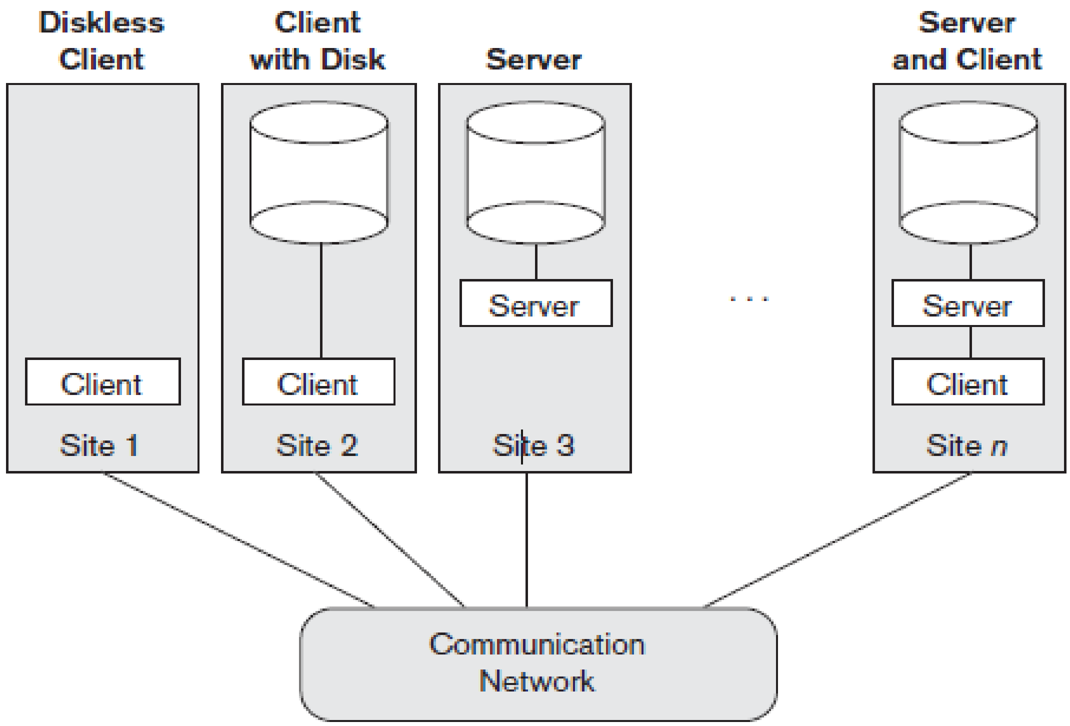
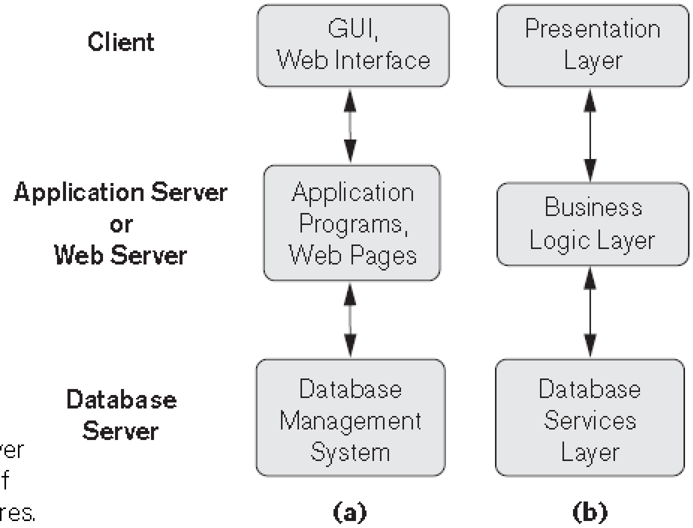


fig: Physical two-tier client/server architecture

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, as illustrated in Figure. This intermediate layer or **middle tier** is called the **application server** or the **Web server**, depending on the application. Clients contain GUI interfaces and some additional application-specific business rules. The intermediate server accepts requests from the client, processes the request and sends



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

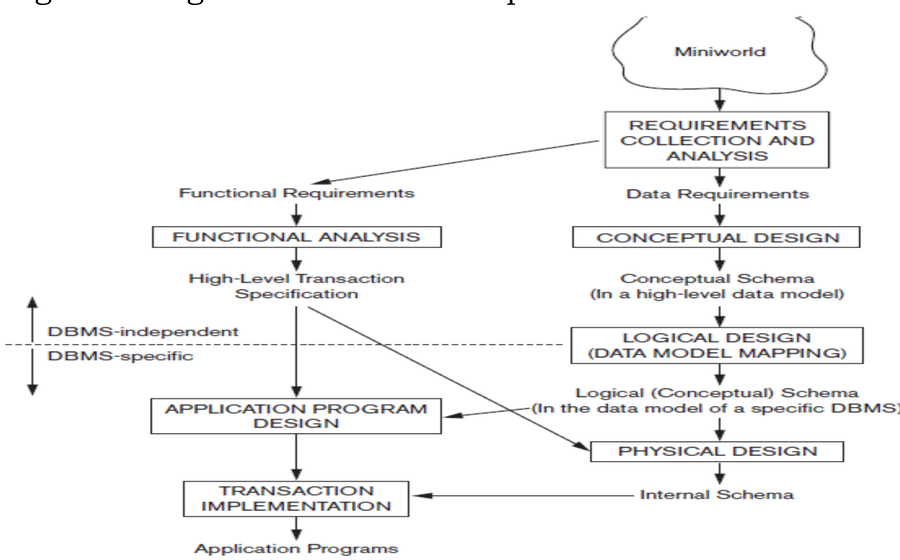
database queries and commands to the database server. the *user interface, application rules,* and *data access* act as the three tiers. Figure. shows another architecture used by database and other application package vendors. The presentation layer displays information to the user and allows data entry. The business logic layer handles intermediate rules and constraints before data is passed up to the user or down to the DBMS. The bottom layer includes all data management services. The middle layer can also act as a Web server, which retrieves query results from the database server and formats them into dynamic Web pages that are viewed by the Web browser at the client side.

Using High-Level Conceptual Data Models for Database Design 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**. 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 next step in database design is the actual implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model—such as the relational or the object-relational database model—so the conceptual schema is transformed from the high-level data model into the implementation data model. This step is called **logical design** or **data model mapping.**

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.

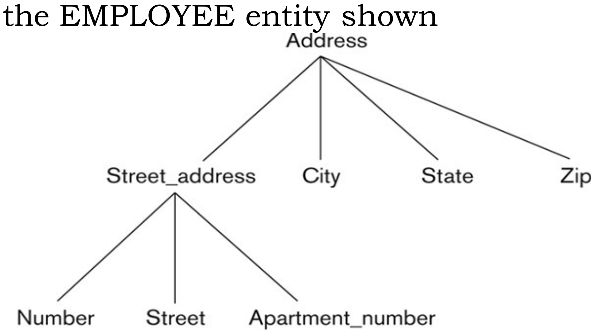


ENTITY: which is a *thing* in the Entity: A real world with an independent existence. An entity may be an object with a physical existence or it may be an object with a conceptual existence. example, a particular person, car, house, or employee.
ATTRIBUTES— the particular properties that describe an entity. For example, an EMPLOYEE entity may be described by the employee's name, age, address, salary, and job.

Types of attributes

1. Composite Attributes
2. Simple (Atomic) Attributes
3. Single-Valued Attributes
4. Multi valued Attributes
5. Stored Attributes
6. Derived Attributes
7. Complex Attributes

1. **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 shown



2. **Simple/Atomic Attributes** :Attributes that are not divisible are called **simple** or **atomic attributes**. Ex: Emp_ID of an Employee

3. **Single valued attributes** :Most attributes have a single value for a particular entity; such attributes are called **single-valued**. Ex: Age is a single-valued attribute of a person

4. **Multi valued attributes** :Most attributes have a multivalue for the same property; such attributes are called **Multivalued**. Ex: color : {red, blue}

5. **Stored attribute :** the value of this type can be stored or entered directly to relative attribute entities. Ex: **Birth Date**

6. **Derived attribute:** the value of this type can be derived from the values of the other relative attribute entities. Ex:AGE attribute can be derived by subtracting the date of DOB from the current DATE .

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

(AddressPhone((Phone(AreaCode,PhoneNumber)},
Address(StreetAddress(Number,Street,ApartmentNumber),
City,State,Zip)) }
complex attribute: AddressPhone.

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

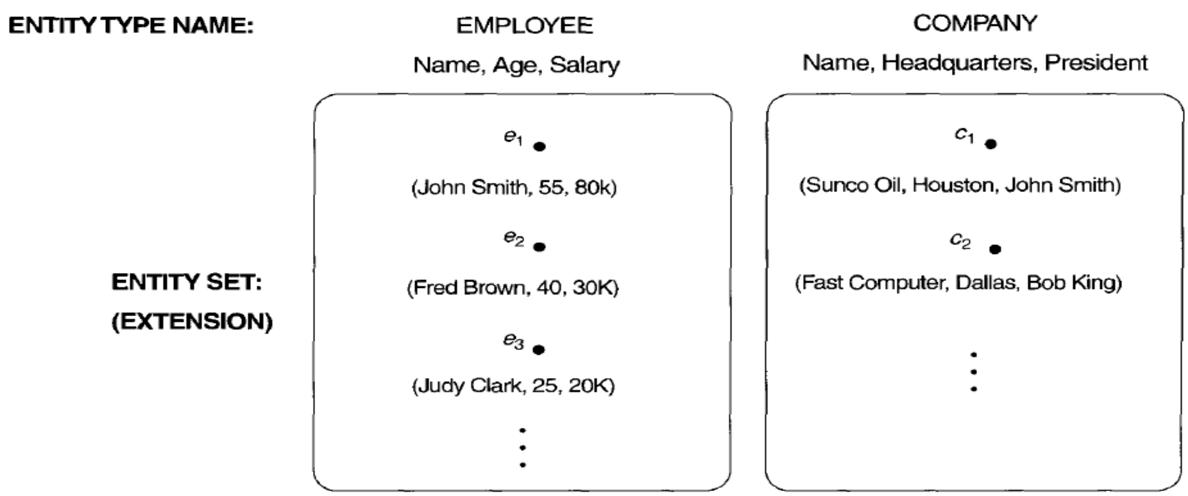
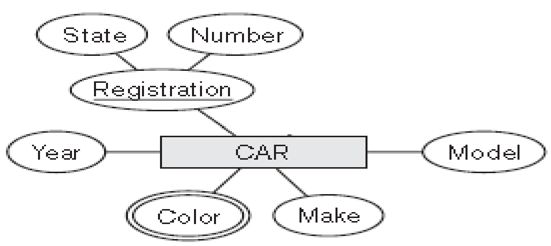


FIGURE Two entity types, EMPLOYEE and COMPANY, and some member entities of each.

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. Each key attribute has its name **underlined** inside the oval.



Entity Set :A entity set is a set of entities of the same type that share the same properties or attributes .

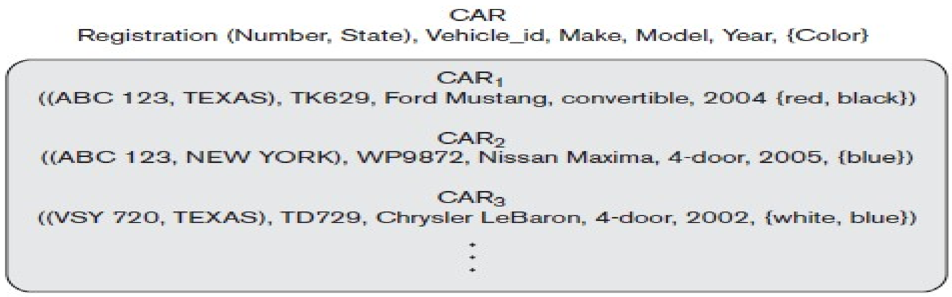


Fig: Entity set with three entities.

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.

Initial Conceptual Design of the COMPANY Database

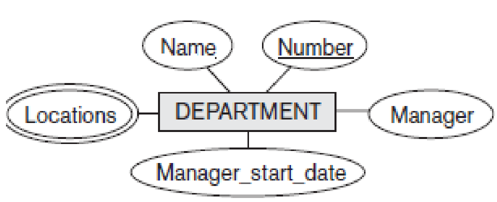
1. Identifying all entity sets
2. Identifying attributes with all entity sets (aware of different attributes)
3. Identifying feasible relationship terms
4. Identifying cardinality ratios
5. Identifying participating constraints
6. Identifying participating roles(if any)

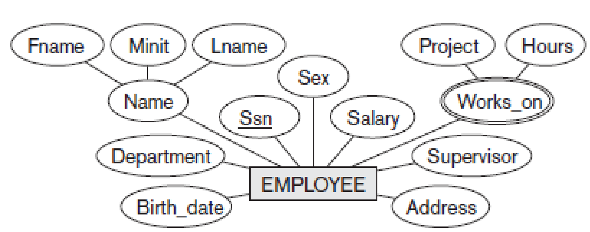
Entity types for the **COMPANY** database. we can identify four entity types—one Corresponding to each of the four items in the specification

1. An entity type DEPARTMENT with attributes Name, Number, Locations, Manager, and Manager_start_date. Locations is the only multivalued attribute.

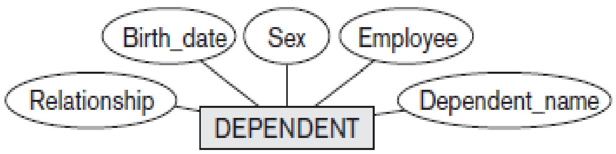
2. An entity type PROJECT with attributes Name, Number, Location, andControlling_department. Both Name and Number are (separate) key attributes.

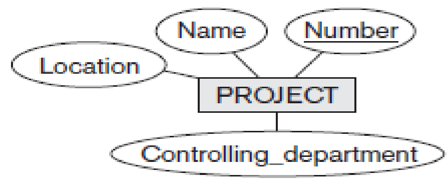
3. An entity type EMPLOYEE with attributes Name, Ssn, Sex, Address, Salary,Birth_date, Department, and Supervisor. Both Name and Address may be composite attributes. components of Name—First_name, Middle_initial, Last_name—or of Address.





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

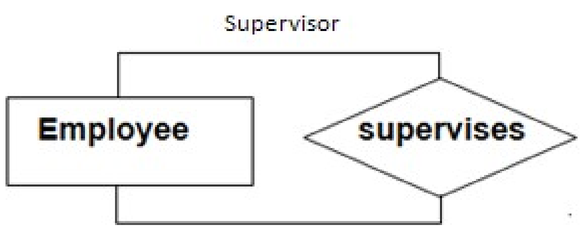




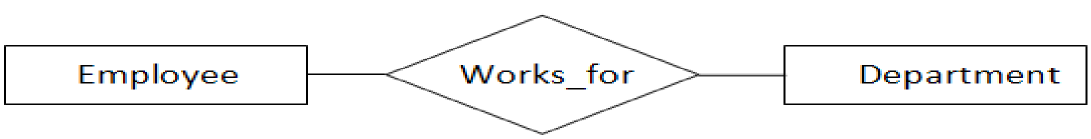
RELATIONSHIPS: A relationship relates two or more distinct entities with a specific meaning OR is an association among entities. Entity does not exists in isolation Ex: EMPLOYEE John *works_on* the Pro-X PROJECT,

RELATIONSHIP TYPES: a relationships' degree indicates the number of associated entities or participants.

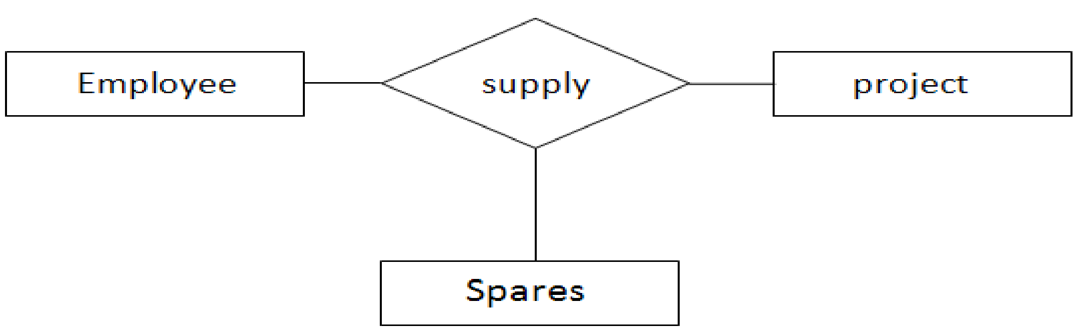
Unary/Recursive Relationship: when an association maintained with a single entity



BINAY RELATIONSHIP: maintained with two entities



TERNARY RELATIONSHIP: maintained with three entities.



Note: although higher exixts,they are not specifically named

RELATIONSHIP SETS: is a set of relationships of the same types. formally ,it is mathematical relation on entity sets. Relationships of the same type are grouped or typed into a relationship type.

For example, the WORKS_ON relationship type in which EMPLOYEEs and PROJECTs participate, or the MANAGES relationship type in which EMPLOYEEs and DEPARTMENTs participate.

• The degree of a relationship type is the number of participating entity types.

– Both MANAGES and WORKS_ON are *binary* relationships.

Constraints on Relationship Types

There are two types of relationship ***constraints***-

1.Cardinality Ratio

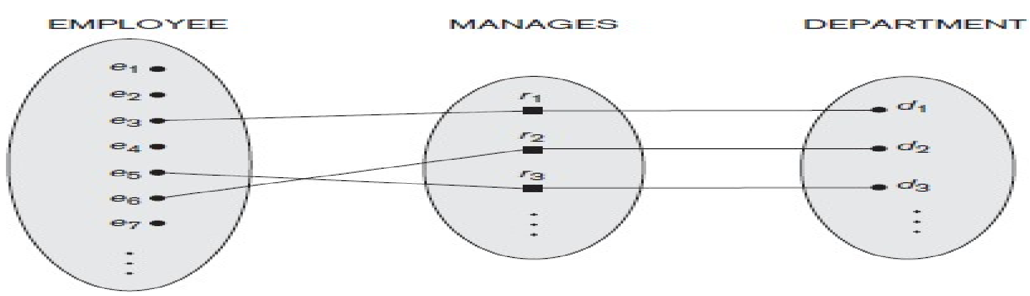
2.Participation Constraint

Cardinality ratio specifies the number of relationship instances that can participate. it is a characteristic of relationships. Common cardinality ratios are-

1. One to One (1:1)
2. One to Many (1:N)
3. Many to One(N:1)
4. Many to Many (M:N)

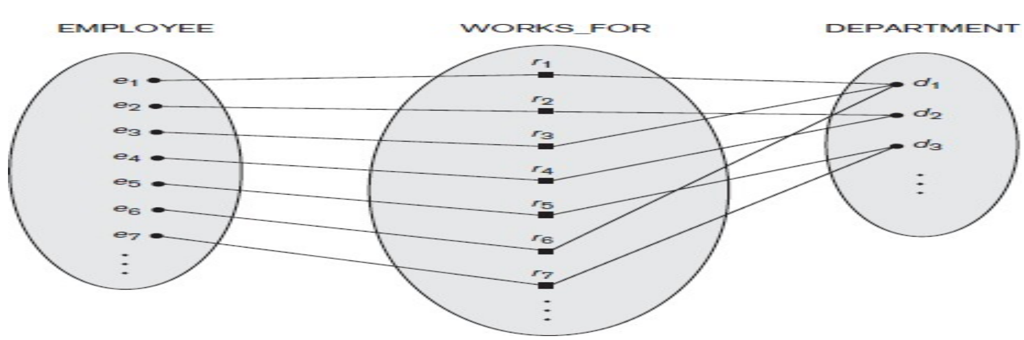
ONE TO ONE (1:1):any entity in A associated with at most one entity in B

Ex: employee manages department



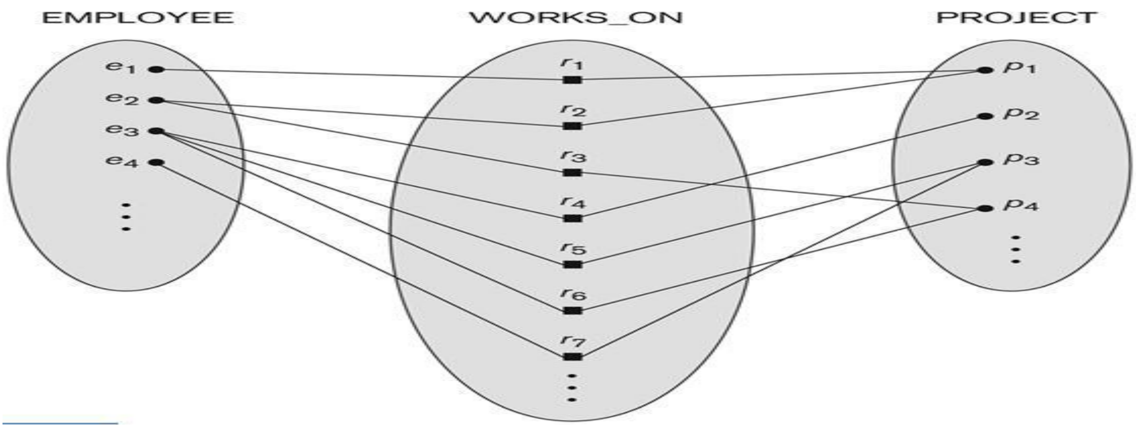
One to Many (1:N):an entity in A associated with any number of entities in B.An entity B, however can be associated with at most one entity in A

Ex: employee worksfor department



Many to one (N:1):any entity in A associated with atmost one entity in B. an entity B can be associated with any number of entities in A
Ex: many Employees worksfor one department
Many to many (M:N):any entity in A associated with any number of entities in B. an entity B can be associated with any number of entities in A

Ex: many Employees workson many project.

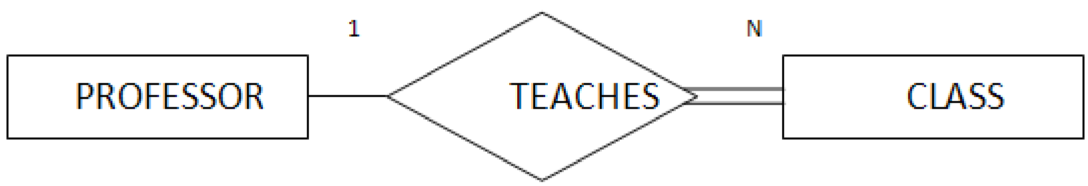


Participating constraints
1. Total participation
2. Partial participation

TOTAL PARTICIPATION : if only if every entities in E participate in relationship R,then the participation of entity entity set E in relationship R is said to be **Total.**

Total participation is also called **existence dependency**.
*The cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type.*

Ex: PROFESSOR Teaches CLASS.



A participating entity in a relation is either *optional or mandatory*. The participation is optional if one entity occurrence does not require a corresponding entity occurrence in a particular relationship . For example ,*some colleges appoints some professors who conduct RESEARCH with out teaching CLASSES , if we examine the relationship "PROFESSOR teaches CLASS ,its quite possible for a professor not to teach a class, there fore ,CLASS is optional to PROFESSOR, on the* *otherhand ,a CLASS must be taught by a PROFESSOR, there fore PROFESSOR is mandatory to CLASS.*

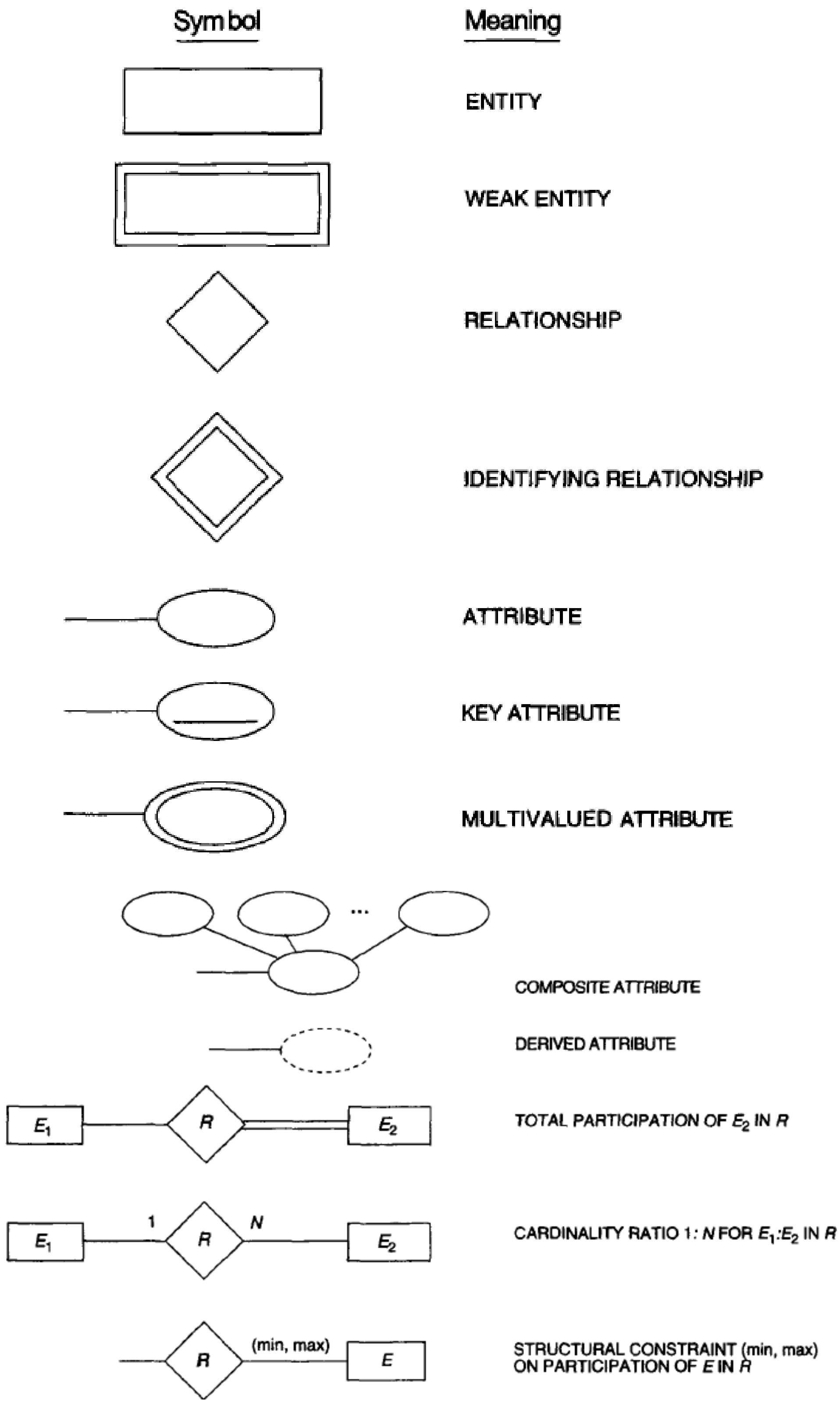
Professor is in **PARTIAL PARTICIPATION**, where as CLASS is in **TOTAL PARTICIPATION.**

Partial participation: if only if some entities in E participate in relationship R,then the participation of entity entity set E in relationship R is said to be **partial.**

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 of its own 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.
A ***weak entity type normally has a partial key***, which is the set of attributes that can uniquely identify weak entities that are *related to the same owner entity.*
Weak entity types can sometimes be represented as complex (composite, multivalued) attributes. The identifying entity type is also sometimes called the parent entity type or the dominant entity type.
The weak entity type is also sometimes called the child entity type or the subordinate entity type.
The partial key is sometimes called the ***discriminator***.

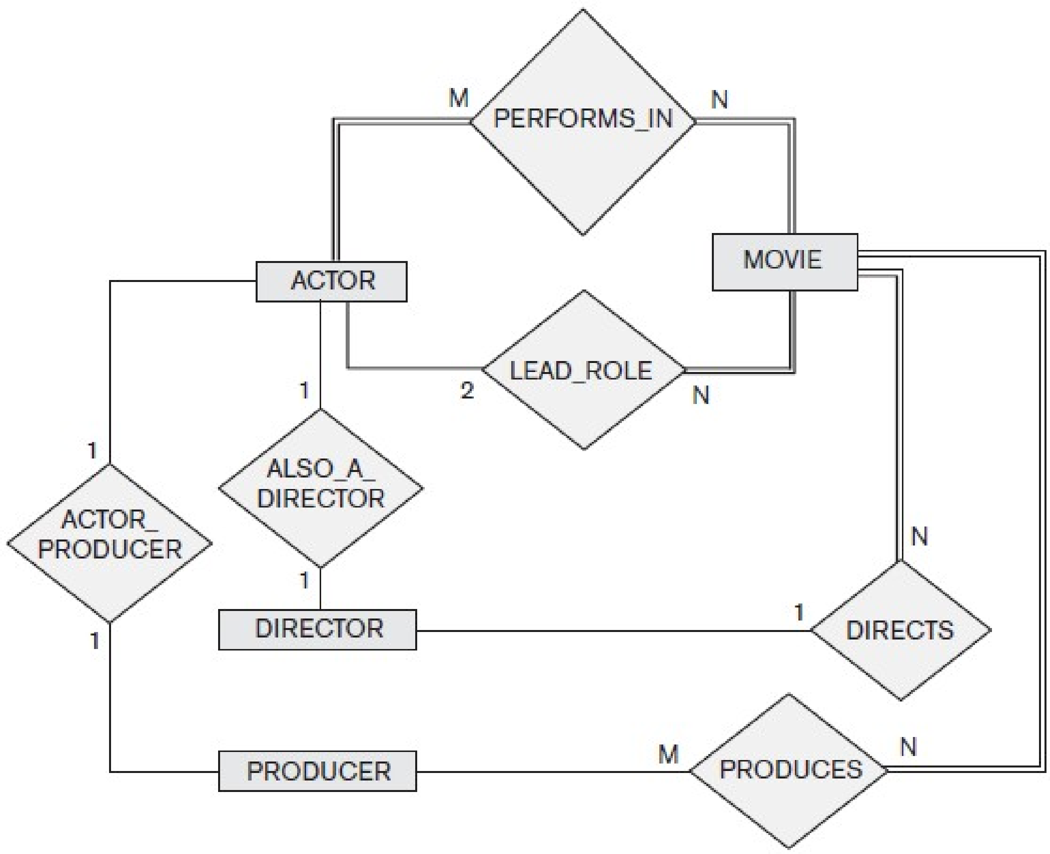
Proper Naming of Schema Constructs
When designing a database schema, the choice of names for entity types, attributes, relationship types, and (particularly) roles is not always straightforward. One should choose names that convey, as much as possible, the meanings attached to the different constructs in the schema. We choose to use *singular names* for entity types, rather than plural ones, because the entity type name applies to each individual entity belonging to that entity type.

ER DIAGRAMS, NAMING CONVENTIONS

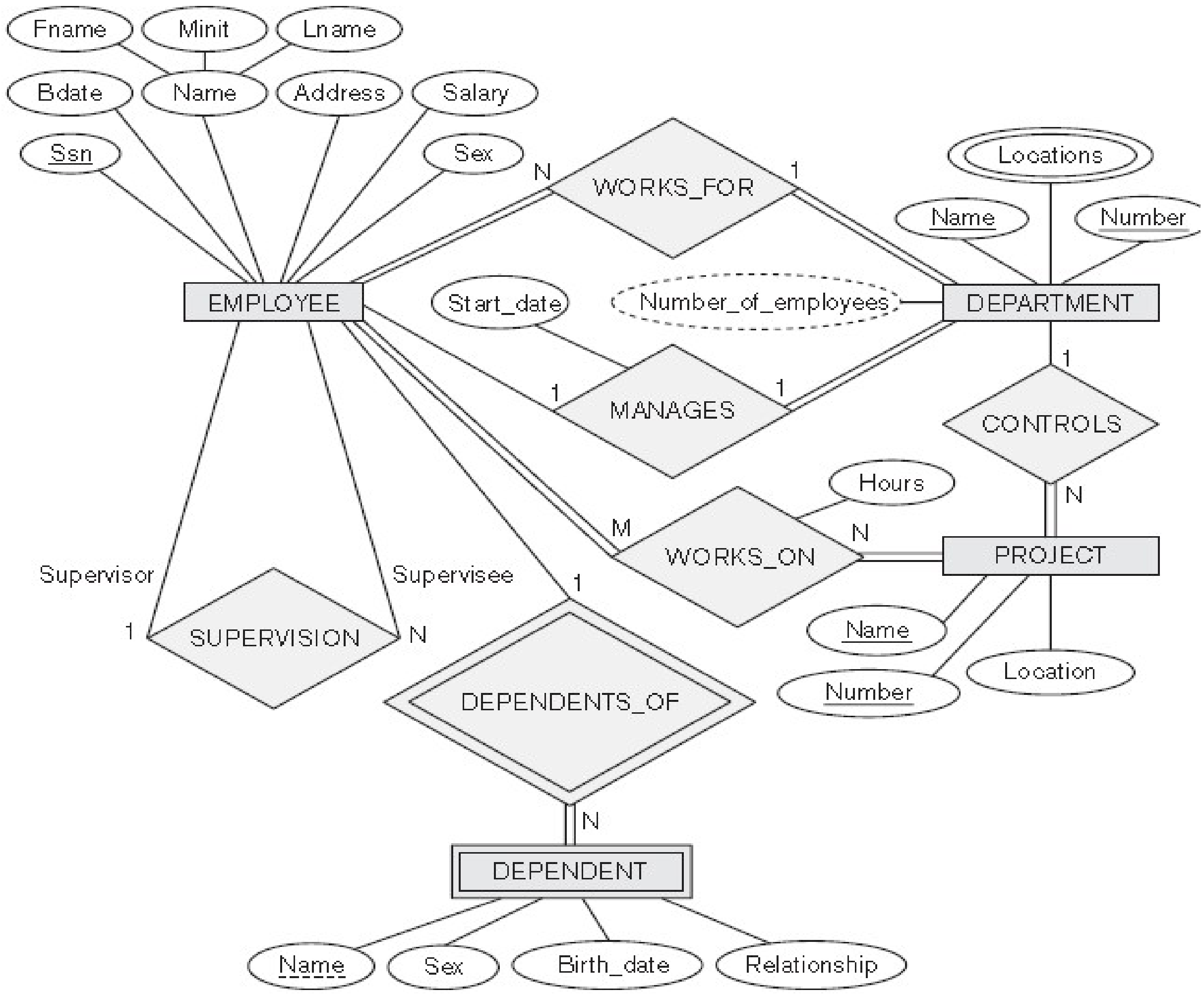


ER DIAGRAMS

MOVIE DATABASE



COMPANY DATABSE/EMPLOYEE DATABASE

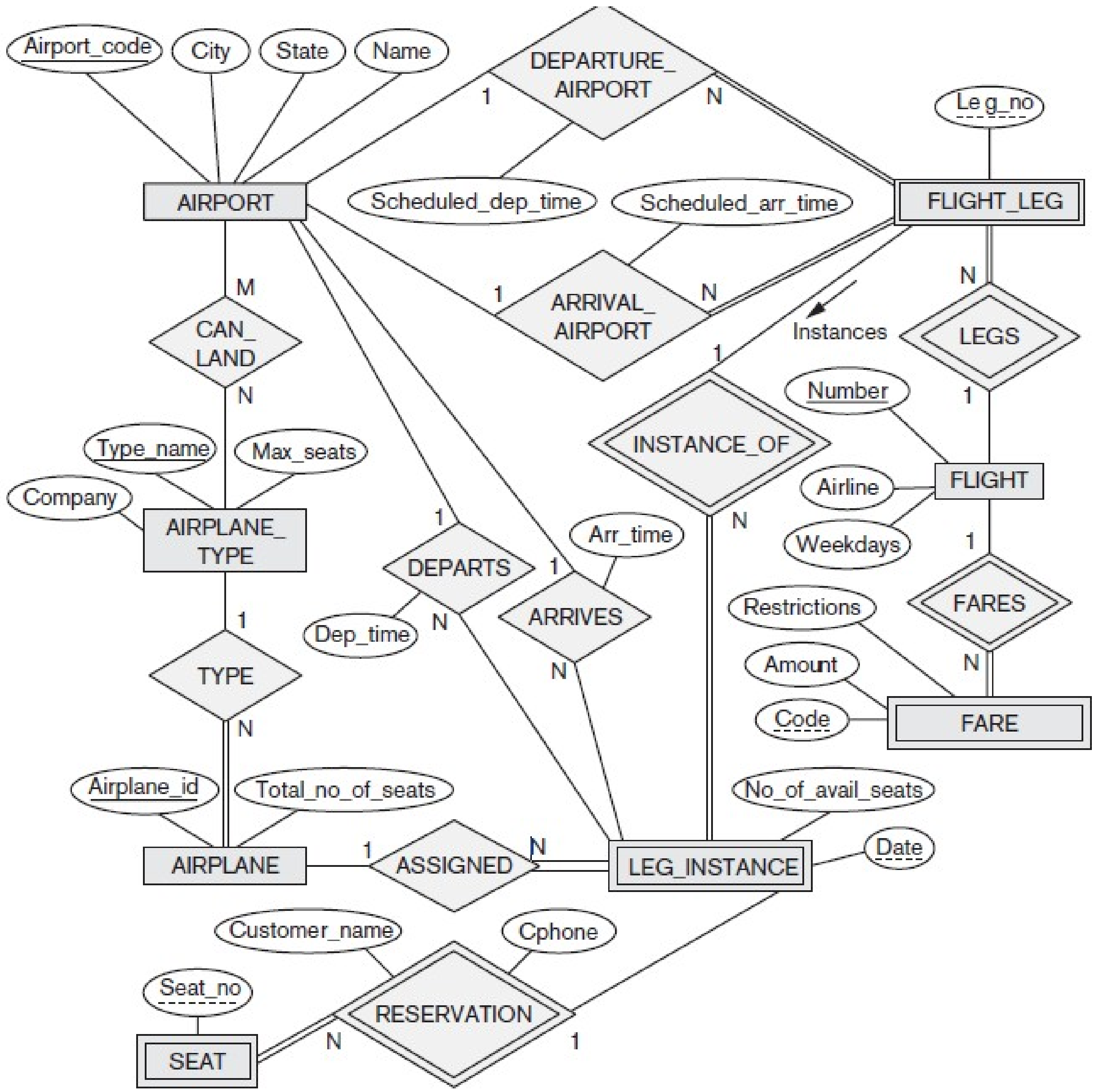


we specify the following relationship types:

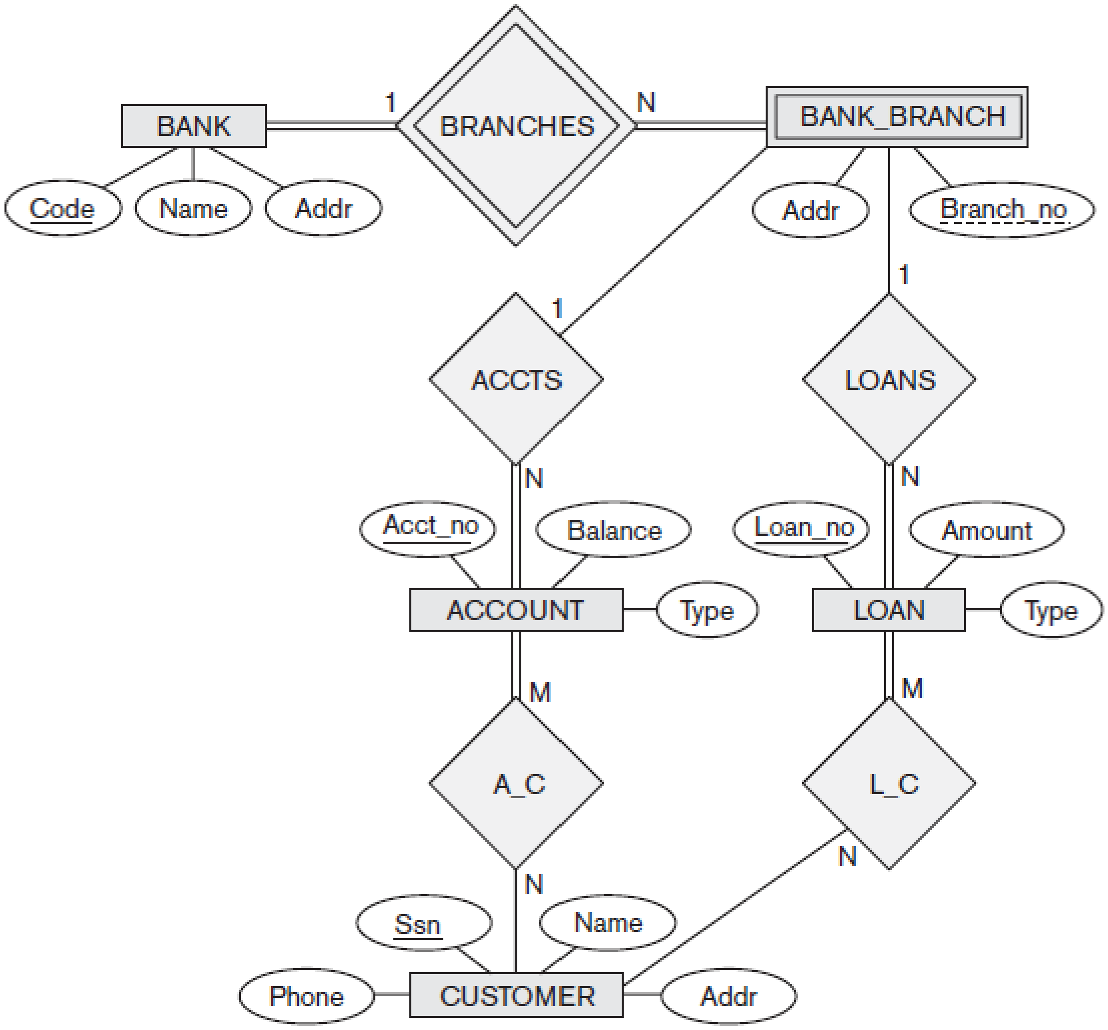
1. MANAGES, a 1:1 relationship type between EMPLOYEE and DEPARTMENT. EMPLOYEE participation ispartial
2. WORKSJOR, a I:N relationship type between DEPARTMENT and EMPLOYEE. Both participations are total.
3. CONTROLS, a I:N relationship type between DEPARTMENT and PROJECT. The participation of PROJECT is total, whereas that of DEPARTMENT is determined to be partial
4. SUPERVISION, a I:N relationship type between EMPLOYEE (in the supervisor role) and EMPLOYEE (in the supervisee role). Both participations are determined to be partial.
5. WORKS_ON, determined to be an M:N relationship type with attribute Hours.

6. DEPENDENTS_OF, a l: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.

AIR LINE DATABASE



BANKING DATABASE



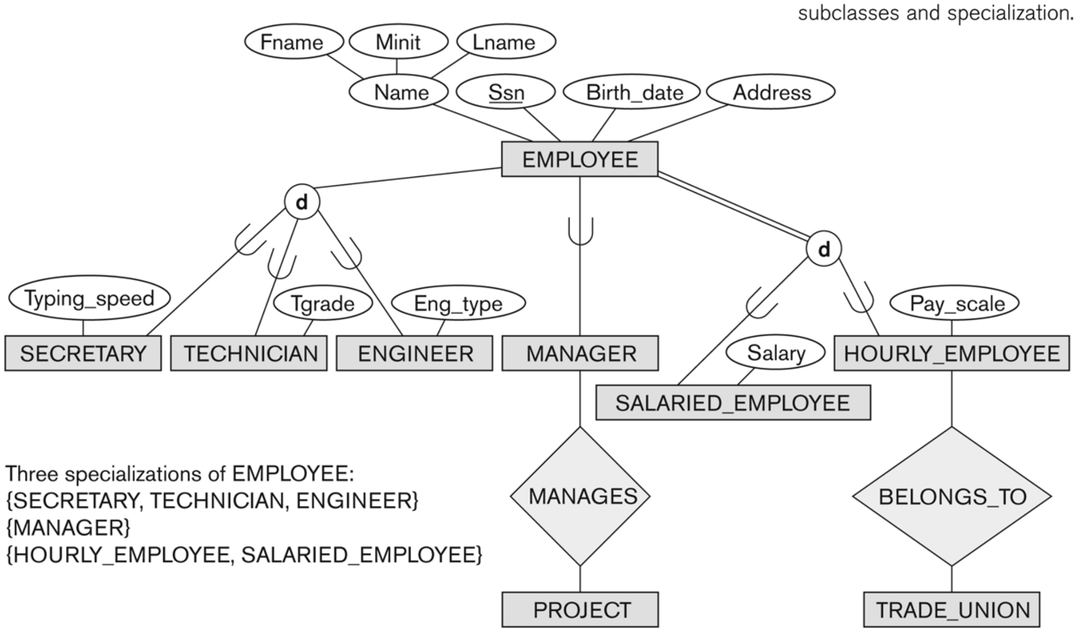
Specialization

Specialization is the process of defining a set of subclasses of a superclass
The set of subclasses is based upon some distinguishing characteristics of the entities in the superclass

Example: {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of EMPLOYEE based upon *job type.*
May have several specializations of the same superclass

Another specialization of EMPLOYEE based on *method of pay* is {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}.

Figure 4.1
EER diagram notation to represent subclasses and specialization.



Generalization

Generalization is the reverse of the specialization process
Several classes with common features are generalized into a superclass;
original classes become its subclasses
Example: CAR, TRUCK generalized into VEHICLE;
both CAR, TRUCK become subclasses of the superclass VEHICLE.
We can view {CAR, TRUCK} as a specialization of VEHICLE
Alternatively, we can view VEHICLE as a generalization of CAR and TRUCK

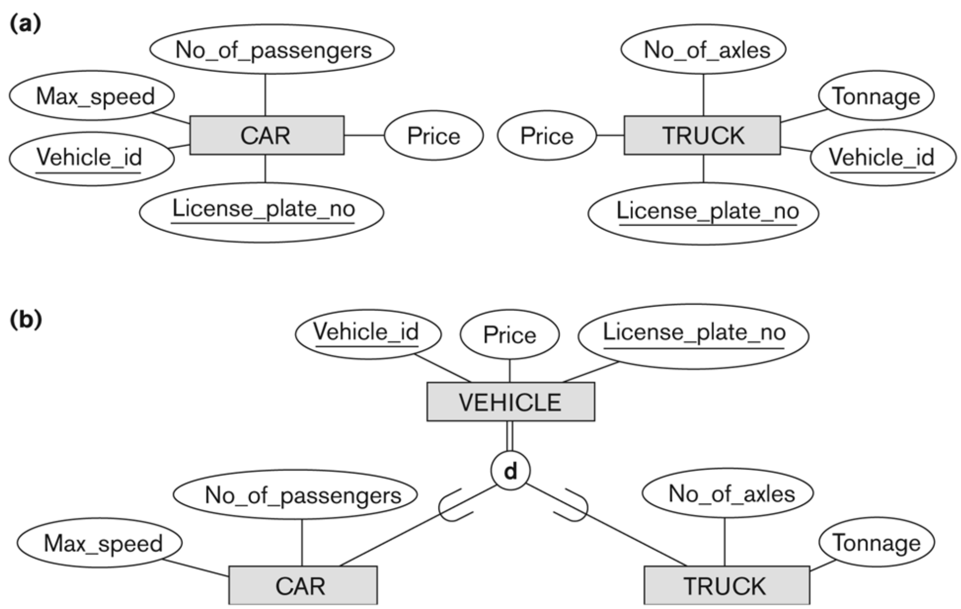


Figure 4.3
Generalization. (a) Two entity types, CAR and TRUCK.
(b) Generalizing CAR and TRUCK into the superclass VEHICLE.

Diagrammatic notation are sometimes used to distinguish between generalization and specialization. Arrow pointing to the generalized superclass represents a generalization. Arrows pointing to the specialized subclasses represent a specialization.
If all subclasses in a specialization have membership condition on same attribute of the superclass, specialization is called an attribute-defined specialization
Attribute is called the defining attribute of the specialization
Example: JobType is the defining attribute of the specialization {SECRETARY, TECHNICIAN, ENGINEER} of EMPLOYEE
If no condition determines membership, the subclass is called user-defined.
Membership in a subclass is determined by the database users by applying an operation to add an entity to the subclass. Membership in the subclass is specified individually for each entity in the superclass by the user.