

Basic Concepts

Introduction:-

Any organization, be it a Bank, Manufacturing Company, Hospital, University, Conglomerate or a Government Department, all require huge amount of data in one form or another. All such organizations need to collect data, process them and store them for future use. These organizations require data for a number of purposes, say:

- Preparing sales report
- Forecasts
- Accounts payable & receivable
- Medical histories, etc.

Thus we can say that data are very vital corporate resources. The amount of data used these days in organizations can be measured in the range of some billions of bytes. The financial investment involved is also considerable.

Traditional databases are organized by fields, records, and files. A field is a single piece of information; a record is one complete set of fields; and a file is a collection of records. For example, a telephone book is analogous to a file. It contains a list of records, each of which consists of three fields: name, address, and telephone number.

Data:

Data are the raw facts that can be recorded and that have some implicit meaning. For example consider the names, telephone numbers and address of people you know. All these names are somewhere linked to persons whom you want to remember or want to keep contact with them hence there is some meaning full reason to why you have collected this data.

Traditional databases are organized by fields, records, and files. A field is a single piece of information; a record is one complete set of fields; and a file is a collection of records. For example, a telephone book is analogous to a file. It contains a list of records, each of which consists of three fields: name, address, and telephone number.

Database:

A database is a organized collection of related data with some implicit meaning. The data collected in a database is stored in a standard format designed to be share by one or multiple users. For example consider the data mentioned above address, names etc. of known persons you may have recorded them in the from of address book, or using a personal computer with software such as WORD or EXCEL etc. This is a collection of related data with an implicit meaning and hence a database. But, any abrupt collection of data cannot be called a database. A database has to have some implicit properties.

- A database should be able to represent some aspect of real world (also called mini world). Changes to mini world are reflected in the database.
- A database is a logically interrelated collection of data with some inherent meaning. A random collection of data thus cannot be referred to as a database.
- A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.

A database can be of any size and of varying complexity. Hence it may be generated and maintained manually or by machine. Hence we can say that any database can be created and maintained in two various ways:

- Manual
- Computerized
 - File Oriented System
 - Data Base 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 hence a *general-purpose software system* that facilitates the processes of *defining*, *constructing*, and *manipulating* databases for various applications.

- **Defining** a database involves specifying the data types, structures, and constraints for the data to be stored in the database.
- **Constructing** the database is the process of storing the data itself on some storage medium that is controlled by the DBMS.
- **Manipulating** a database includes such functions as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data.

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

Database system

We will call the database and DBMS software together a **database system**.

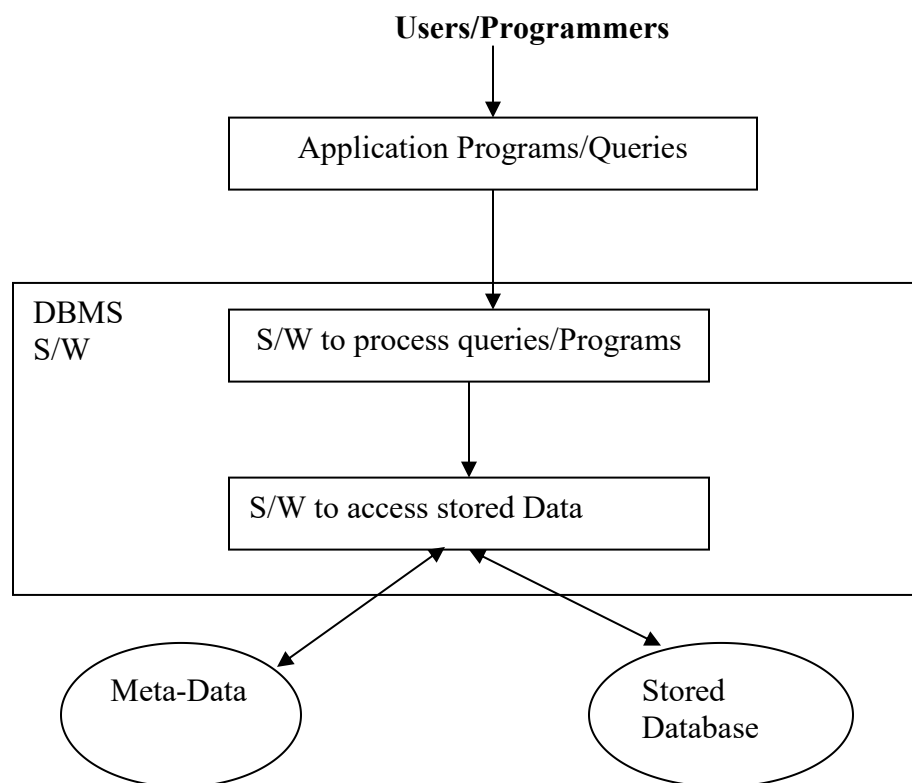


Fig.: A Database System Environment

Database example:

Let us consider an example that most reader familiar with: ‘Any UNIVERSITY Database’ for maintaining information containing students, courses, and grades in a university environment. In the fig. below, the database is organized as five files, each of which stores data records of the same type.

Student file stores data on each student. **Course** file stores data on each course.

Section file stores data on each section of a course.

Grade-Report file stores the grades that student receive in the various sections they have completed.

Prerequisite file stores the prerequisite of each course.

STUDENT	Name	Roll No	Class	Major
	Amit Kr	17	B.Cam	Comp. Sc
	Reha	8	B.com	Marketing

COURSE	Course Name	Course No.	Hours	Department
	Intro to IT	101	4	Comp Sc
	Data Structure	102	4	Comp Sc
	Mathematics	103	3	Math
	DBMS	104	3	Comp Sc

SECTION	Section Identifier	Course No.	Semester	Year	Instructor
	A	103	I	91	Mrs. A. Panigrahi
	B	101	I	91	Mr. P Kumar
	C	102	II	92	Mr. A. Sinha
	D	103	I	92	Mrs. P Agg.
	E	101	I	92	Mr. P Kumar
	F	104	I	92	Mr. S Kumar

GRADE_REPORT	Student No.	Section Identifier	Grade
	17	D	β
	17	E	γ
	8	A	α
	8	B	α
	8	C	β
	8	F	α

PREREQUISITE	Course No.	Prerequisite No.
	104	102
	104	103
	102	101

Fig : Example of a database

Database System Vs File systems:

Basis	DBMS Approach	File System Approach
Meaning	DBMS is a collection of data. In DBMS, the user is not required to write the procedures.	The file system is a collection of data. In this system, the user has to write the procedures for managing the database.
Sharing of data	Due to the centralized approach, data sharing is easy.	Data is distributed in many files, and it may be of different formats, so it isn't easy to share data.
Data Abstraction	DBMS gives an abstract view of data that hides the details.	The file system provides the detail of the data representation and storage of data.
Security and Protection	DBMS provides a good protection mechanism.	It isn't easy to protect a file under the file system.
Recovery Mechanism	DBMS provides a crash recovery mechanism, i.e., DBMS protects the user from system failure.	The file system doesn't have a crash mechanism, i.e., if the system crashes while entering some data, then the content of the file will be lost.
Manipulation Techniques	DBMS contains a wide variety of sophisticated techniques to store and retrieve the data.	The file system can't efficiently store and retrieve the data.
Concurrency Problems	DBMS takes care of Concurrent access of data using some form of locking.	Concurrent access has many problems like redirecting the file while deleting some information or updating some information.
Where to use	Database approach used in large systems which interrelate many files.	File system approach used in large systems which interrelate many files.
Cost	The database system is expensive to design.	The file system approach is cheaper to design.
Data Redundancy & Inconsistency	Due to the centralization of the database, the problems of data redundancy and inconsistency are controlled.	The files and application programs are created by different programmers so there exists a lot of duplication of data which may lead to inconsistency.
Structure	The database structure is complex to design.	The file system approach has a simple structure.
Data Independence	In this system, Data Independence exists, and it can be of two types. <ul style="list-style-type: none"> • Logical Data Independence • Physical Data Independence 	In the File system approach, there exists no Data Independence.
Integrity Constraints	Integrity Constraints are easy to apply.	Integrity Constraints are difficult to implement in file system.
Data Models	In the database approach, 3 types of data models exist: <ul style="list-style-type: none"> • Hierarchical data models • Network data models • Relational data models 	In the file system approach, there is no concept of data models exists.

Flexibility	Changes are often a necessity to the content of the data stored in any system, and these changes are more easily with a database approach.	The flexibility of the system is less as compared to the DBMS approach.
Examples	Oracle, SQL Server etc.	C, C++ , Java etc.

Persons involved in Database System

For a small personal database, one person typically defines, constructs, and manipulates the database. However, many persons are involved in the design, use, and maintenance of a large database with a few hundred users. In this section we identify the people whose jobs involve the day-to-day use of a large database; we call them the "actors on the scene."

1. **Database Designer:** A database designer (DD) is the person who is responsible for designing the actual database. The DD should interact with all the potential groups of users and develop an External as well as Logical view of the database that meets the

Responsibilities:

- a) Identifying the data to be stored in the database
 - b) Choosing appropriate structure to represent and store this data. This is done before the database is actually implemented.
 - c) It is the responsibility of DD to communicate with all the prospective database users, in order to understand their requirements.
 - d) The DD should come up with a design of the database, which should meet end user requirements and should be capable enough to perform all data processing functions.
2. **Database Administrator (DBA):** In an organization where many persons are using the same resources, there is a need of chief administrator to oversee and manage these resources. In a database environment the primary resource is the database itself and secondary resource is the DBMS and related software. Administering these resources is the responsibility of the database administrator (DBA).

Responsibilities:

- ❖ **Schema Definition:**
- ❖ **Storage Structure and Access-method Definition:**
- ❖ **Schema Modification:**
- ❖ **Data Access authorization granting:**
- ❖ **Integrity-Constraint specification:**
- ❖ **Data Appraisals:**
- ❖ **Preparing Data Manuals:**

3. **End Users**

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

4. **System Analysts and Application Programmers (Software Engineers)**

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

Application of Database System:

Databases are widely used. Here are some representative applications:

1. **Banking:** For customer information, accounts, and loans, and banking transactions.
2. **Airlines:** For reservations and schedule information. Airlines were among the first to use database in a geographically distributed manner—Terminals situated around the world accessed the central database system through phone lines and other data networks.
3. **Universities:** For student information, course registrations, and grades.
4. **Credit card transaction:** For purchases on credit cards and generation of monthly statements.
5. **Telecommunication:** For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.
6. **Finance:** For storing information about holdings, sales, and purchases of financial instrument such as stocks and bonds.
7. **Sales:** For customer, product, and purchase information.
8. **Manufacturing:** For management of supply chain and for tracking production of item in factories, inventories of items in warehouses/stores, and orders for items.
9. **Human resources:** For information about employees, salaries, payroll taxes and benefits, and for generation of paychecks.

Data Models, Schemas, and Instances

One fundamental characteristic of the database approach is that it provides some level of data abstraction by hiding details of data storage that are not needed by most database users.

Data Model

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

Categories of Data Models

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

1. **High-level (or conceptual/ Object-based logical models) data models** provide concepts that are close to the way many users perceive data. One of the good examples is ER model that contains concepts like entity, attribute, and relationship. An **entity** represents a real-world object or concept, such as an employee or a project that is described in the database. An **attribute** represents some property of interest that further describes an entity, such as the employee's name or salary. A **relationship** among two or more entities represents an interaction among the entities; for example, a works-on relationship between an employee and a project. Other examples are: object oriented model, semantic data model, functional data model.
2. **Representational (or implementation/ Record-based logical models) data models**, which provide concepts that may be understood by end users but that are not too far removed from the way data is organized within the computer. Representational data models hide some details of data storage but can be implemented on a computer system in a direct way.

Representational or implementation data models are the models used most frequently in traditional commercial DBMSs, and they include the widely-used **relational data model**, as well as the **network** and **hierarchical models**—that have been widely used in the past.

3. **Low-level (or physical) data models** provide concepts that describe the details of how data is stored in the computer. Concepts provided by low-level data models are generally meant for computer specialists, not for typical end users. Physical data models describe how data is stored 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.

Schemas, Instances, and Database State

The description of a database is called the **database schema**, which is specified during database design and is not expected to change frequently. Most data models have certain conventions for displaying the schemas as diagrams. A displayed schema is called a **schema diagram**.

Figure below shows a schema diagram for the database;

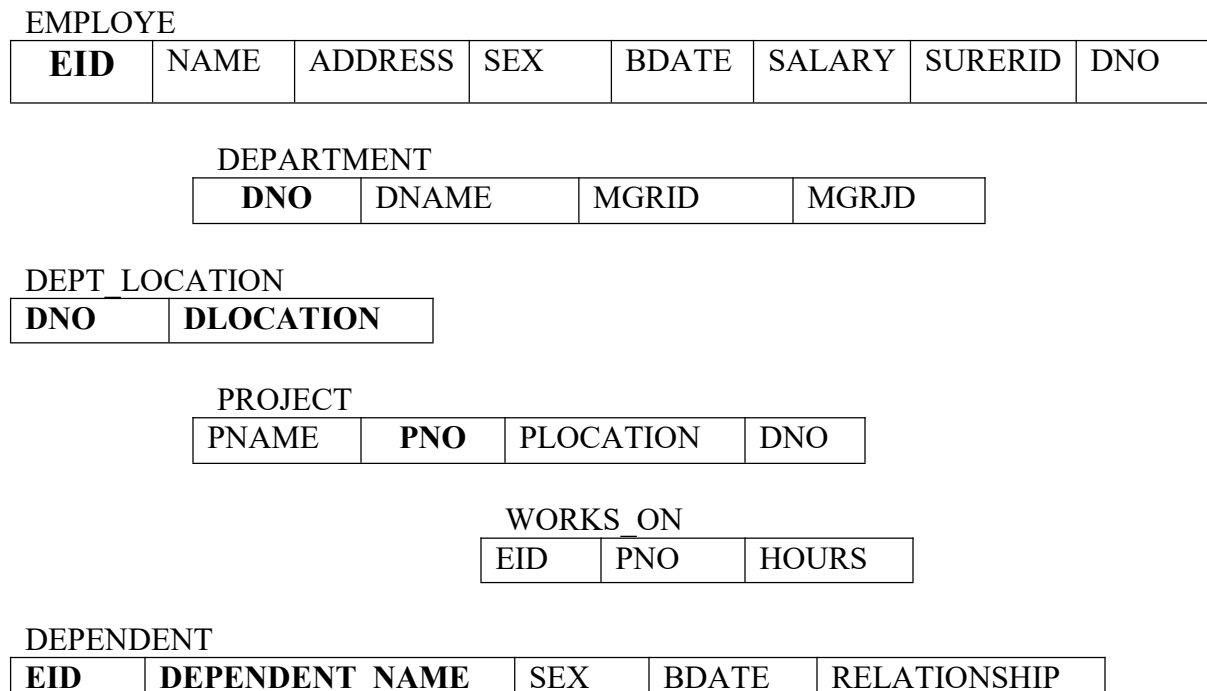


Fig.- Schema diagram of the COMPANY database

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. Many database states can be constructed to correspond to a particular database schema. Every time we insert or delete a record, or change the value of a data item in a record, we change one state of the database into another state.

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 stores the descriptions of the schema constructs and constraints—also called the **meta-data**—in the DBMS catalog.

DBMS Architecture and Data Independence

The important characteristics of the database approach, described earlier, **three-schema architecture**, which was proposed to help achieve and visualize these characteristics.

The Three-Schema Architecture

The goal of the three-schema architecture is to separate the user applications and 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. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
2. The **conceptual level** has a **conceptual schema**, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.
3. The **external or view level** includes a number of **external schemas** or **user views**. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

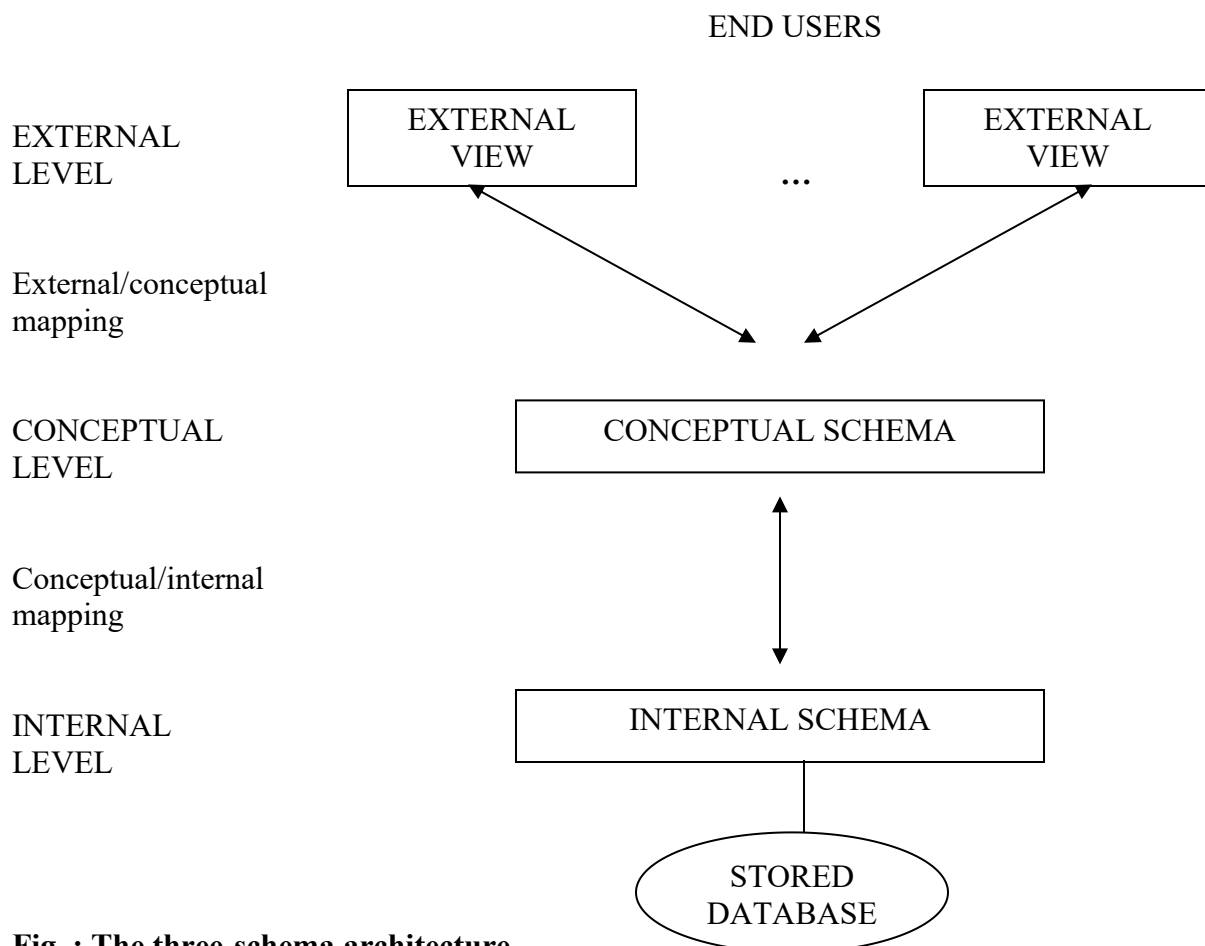


Fig. : The three-schema architecture

The three-schema architecture is a convenient tool for the user to visualize the schema levels in a database system. Most DBMSs do not separate the three levels completely, but support the three-

schema architecture to some extent. Some DBMSs may include physical-level details in the conceptual schema. In most DBMSs that support user views, external schemas are specified in the same data model that describes the conceptual-level information. Some DBMSs allow different data models to be used at the conceptual and external levels.

Notice that the three schemas are only *descriptions* of data; the only data that *actually* exists is at the physical level. In a DBMS based on the three-schema architecture, each user group refers only 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 a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The processes of transforming requests and results between levels are called **mappings**. These mappings may be time-consuming, so some DBMSs—especially those that are meant to support small databases—do not support external views. Even in such systems, however, a certain amount of mapping is necessary to transform requests between the conceptual and internal levels.

Data Independence

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

1. **Logical data independence**
2. **Physical 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), or to reduce the database (by removing a record type or data item). In the latter case, external schemas that refer only to the remaining data should not be affected. Only the view definition and the mappings need be changed in a DBMS that supports logical data independence. Application programs that reference the external schema constructs must work as before, after the conceptual schema undergoes a logical reorganization. Changes to constraints can be applied also to the conceptual schema without affecting the external schemas or application programs.

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

Whenever we have a multiple-level DBMS, its catalog must be expanded to include information on how to map requests and data among the various levels. The DBMS uses additional software to accomplish these mappings by referring to the mapping information in the catalog. Data independence is accomplished because, when the schema is changed at some level, the schema at the next higher level remains unchanged; only the *mapping* between the two levels is changed. Hence, application programs referring to the higher-level schema need not be changed.

The three-schema architecture can make it easier to achieve true data independence, both physical and logical. However, the two levels of mappings create an overhead during compilation or execution of a query or program, leading to inefficiencies in the DBMS. Because of this, few DBMSs have implemented the full three-schema architecture.

Database Languages

The DBMS must provide appropriate languages and interfaces for each category of users. In this section we discuss the types of languages and interfaces provided by a DBMS and the user categories targeted by each interface.

DBMS Languages

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

Data Definition Language (DDL): A database schema is specified by a set of definitions expressed by a special language called a data definition language (DDL). The result of compilation of DDL statements is a set of tables that is stored in a special file called data dictionary, or data directory. A data dictionary is a file that contains metadata -that is, data about data.

The basic functions of DDL is to take care of following:

- The schema for each table
- The set of values associated with each attribute
- The integrity constraints
- The set of indices to be maintained for each table
- The security and authorization information for each table
- The physical storage structure of each table on disk

Storage definition language (SDL):

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 (VDL):

For a true three-schema architecture, we would need a third language, the **view definition language (VDL)**, to specify user views and their mappings to the conceptual schema, but in most DBMSs the DDL is used to define both conceptual and external schemas.

Data manipulation language (DML):

Once the database schemas are compiled and the database is populated with data, users must have some means to manipulate the database. Typical manipulations include retrieval, insertion, deletion, and modification of the data. The DBMS provides a **data manipulation language (DML)** for these purposes. There are two main types of DMLs.

A **high-level** or **nonprocedural DML** can be used on its own to specify complex database operations in a concise manner. Many DBMSs allow high-level DML statements either to be entered interactively from a terminal (or monitor) or to be embedded in a general-purpose programming language. In the latter case, DML statements must be identified within the program so that they can be extracted by a pre-compiler and processed by the DBMS.

A **low-level** or **procedural DML** *must* be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects from the database and processes each separately. Hence, it needs to use programming language constructs, such as looping, to retrieve and process each record from a set of records. Low-level DMLs are also called **record-at-a-time DMLs** because of this property. High-level DMLs, such as SQL, can specify and retrieve many records in a single DML

statement and are hence called **set-at-a-time** or **set-oriented DMLs**. A query in a high-level DML often specifies *which* data to retrieve rather than *how* to retrieve it; hence, such languages are also called **declarative**.

Data Control Language (DCL): DCL is a language, which is used to impose features and thus prevents unauthorized access to data in the database. Security is provided by granting or revoking privileges on a user. Privilege determines whether or not a user can execute a given command or a command can be executed on specific groups of data.

In current DBMSs, the preceding types of languages are usually *not considered distinct languages*; rather, a comprehensive integrated language is used that includes constructs for conceptual schema definition, view definition, and data manipulation. Storage definition is typically kept separate, since it is used for defining physical storage structures to fine-tune the performance of the database system, and it is usually utilized by the DBA staff.

A typical example of a comprehensive database language is the SQL relational database language, which represents a combination of DDL, VDL, and DML, as well as statements for constraint specification and schema evolution. The SDL was a component in earlier versions of SQL but has been removed from the language to keep it at the conceptual and external levels only.

Entity Relationship (ER) Diagram

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

Consider an example called COMPANY database that keeps track of a company's employees, departments, and projects.

- 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, and birth date.
- An employee is assigned to one department but may work on several projects, which are not necessarily controlled by the same department.
- We keep track of the number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee.
- 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.

Entities and Attributes

Entity: an **entity**, which is a "thing" in the real world with an independent existence. An entity may be an object with a physical existence—a particular person, car, house, or employee—or it may be an object with a conceptual existence—a company, a job, or a university course etc. It is denoted by rectangle (see fig below)

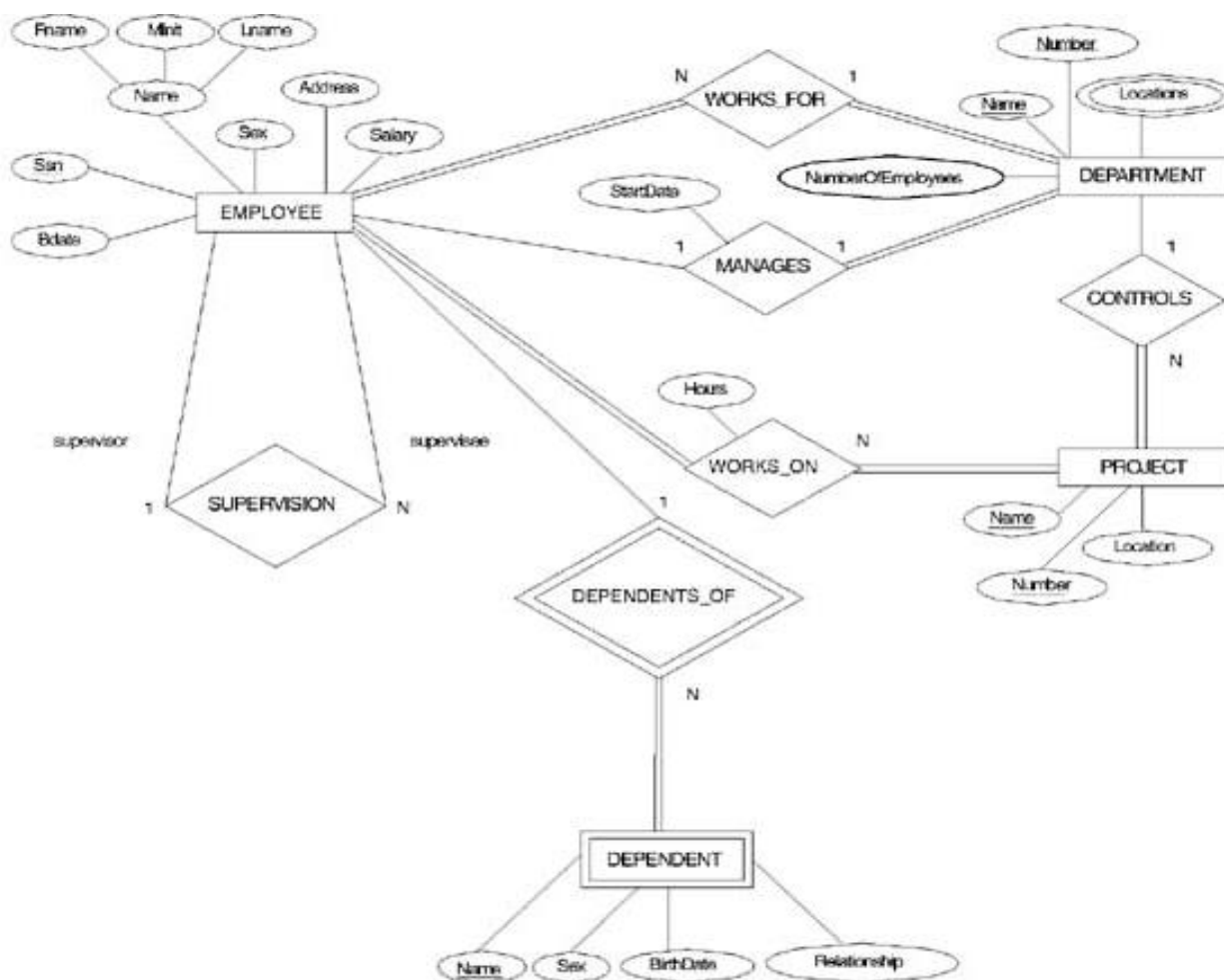


Fig.: : An ER schema diagram for company database

Attribute: 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. An attribute is denoted by oval (see fig above)

Types of Attribute:

Several types of attributes occur in the ER model: *simple* versus *composite*; *single-valued* versus *multivalued*; and *stored* versus *derived*. We first define these attribute types and illustrate their use via examples. We then introduce the concept of a *null value* for an attribute.

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 StreetAddress, City, State, and Pin, with the values "234 Pkt-B," "Dwarka," "N. Delhi," and "110075."

Attributes that are not divisible are called simple or atomic attributes. Composite attributes can form a hierarchy; for example, StreetAddress can be subdivided into three simple attributes, Number, Street, and Apartment Number. The value of a composite attribute is the concatenation of the values of its constituent simple 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 person.

In some cases an attribute can have a set of values for the same entity—for example, a person can have more than one contact numbers or a Colors attribute for a car etc. Multi-valued attributes are denoted by double oval.

Stored Versus Derived Attributes

In some cases two (or more) attribute values are related—for example, the Age and BirthDate attributes of a person. For a particular person entity, the value of Age can be determined from the current (today's) date and the value of that person's BirthDate.

The Age attribute is hence called a derived attribute and is said to be derivable from the BirthDate attribute, which is called a stored attribute.

Null Attributes/Values

Null means no value/blank/nothing. In some cases a particular entity may not have an applicable value for an attribute. For example, the ApartmentNumber attribute of an address applies only to addresses that are in apartment buildings and not to other types of residences, such as single-family homes. Null can also be used if we do not know the value of an attribute for a particular entity. For example, if the HomePhone attribute of a person is null.

Entity Types, Entity Sets, Keys, and Value Sets

Entity Types/Entity Sets

A database usually contains groups of entities that are similar. For example, a company employing hundreds of employees may want to store similar information concerning each of the employees. These employee entities share the same attributes, but each entity has *its own value(s)* for each attribute.

An entity type defines a *collection* (or *set*) of entities that have the same attributes. Each entity type in the database is described by its name and attributes. The collection of all entities of a particular entity type in the database at any point in time is called an entity set; the entity set is usually referred to using the same name as the entity type. For example, EMPLOYEE refers to both a *type of entity* as well as the current *set of all employee entities* in the database.

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

Key Attributes of an Entity Type

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

For example, the EMPLOYEE entity type, a typical key attribute is Employee ID (EID). Sometimes, several attributes together form a key, meaning that the *combination* of the attribute values must be distinct for each entity. If a set of attributes possesses this property, we can define a

composite attribute that becomes a key attribute of the entity type. In ER diagrammatic notation, each key attribute has its name underlined inside the oval.

An entity type may also have *no key*, in which case it is called a **weak entity type**. An entity set that has a key attribute is termed as a **Strong 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.

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

Initial Conceptual Design of the COMPANY Database

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

- An entity type DEPARTMENT with attributes Name, Number, Locations, Manager, and ManagerStartDate. 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.
- An entity type PROJECT with attributes Name, Number, Location, and ControllingDepartment. Both Name and Number are (separate) key attributes.
- An entity type EMPLOYEE with attributes Name, SSN (for social security number), Sex, Address, Salary, BirthDate, 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—FirstName, MiddleInitial, LastName—or of Address.
- An entity type DEPENDENT with attributes Employee, DependentName, Sex, BirthDate, and Relationship (to the employee).

So far, we have not represented the fact that an employee can work on several projects, nor have we represented the number of hours per week an employee works on each project. This characteristic can be represented by a multivalued composite attribute of EMPLOYEE called WorksOn with simple components (Project, Hours).

Relationships, Relationship Types, Roles, and Structural Constraints

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

- The attribute Manager of DEPARTMENT refers to an employee who manages the department;
- The attribute Controlling Department of PROJECT refers to the department that controls the project;
- The attribute Supervisor of EMPLOYEE refers to another employee (the one who supervises this employee);
- The attribute Department of EMPLOYEE refers to the department for which the employee works; and so on.

In the ER model, these references should not be represented as attributes but as relationships. The COMPANY database schema will be refined to represent relationships explicitly. In the initial design of entity types, relationships are typically captured in the form of attributes. As the design is refined, these attributes get converted into relationships between entity types.

Relationship Types/Sets and Instances

A relationship type R among n entity types E_1, E_2, \dots, E_n defines a set of associations—or a relationship set—among entities from these types. As for entity types, a relationship type /relationship set are customarily referred to by the *same name* R .

For example, consider a relationship type WORKS_FOR between the two entity types EMPLOYEE and DEPARTMENT, which associates each employee with the department the employee works for. Each relationship instance in the relationship set WORKS_FOR associates one employee entity and one department entity.

For example, employees e_1, e_3 , and e_6 work for department d_1 ; e_2 and e_4 work for d_2 ; and e_5 and e_7 work for d_3 .

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

Relationship Degree, Role Names, and Recursive Relationships

Degree of a Relationship Type

The degree of a relationship type is the number of participating entity types. Hence, the WORKS_FOR relationship is of degree two. A relationship type of degree two is called binary, and one of degree three is called ternary. Relationships can generally be of any degree, but the ones most common are binary relationships. Higher-degree relationships are generally more complex than binary relationships, and we shall characterize them later.

Role Names and Recursive Relationships

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

For example,

- in the WORKS_FOR relationship type, EMPLOYEE plays the role of *employee* or *worker* and DEPARTMENT plays the role of *department* or *employer*. Role names are not technically necessary in relationship types where all the participating entity types are distinct, since each entity type name can be used as the role name.
- However, in some cases the *same* entity type participates more than once in a relationship type in *different roles*. In such cases the role name becomes essential for distinguishing the meaning of each participation. Such relationship types are called **recursive relationships**. The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity type. Hence, the EMPLOYEE entity type *participates twice* in SUPERVISION: once in the role of *supervisor* (or boss), and once in the role of *supervisee* (or subordinate). Each relationship instance r_i in SUPERVISION associates two employee entities e_j and e_k , one of which plays the role of supervisor and the other the role of supervisee. In ERD Fig the lines marked "1" represent the supervisor role, and those marked "2" represent the supervisee role; hence, e_1 supervises e_2 and e_3 ; e_4 supervises e_6 and e_7 ; and e_5 supervises e_1 and e_4 .

Constraints on Relationship Types

Relationship types usually have certain constraints that limit the possible combinations of entities that may participate in the corresponding relationship set. For example, if the company has a rule

that each employee must work for exactly one department, then we would like to describe this constraint in the schema. We can distinguish two main types of relationship constraints:

- Cardinality ratio (or Mapping cardinality)
- Participation constraint (or existence dependency).

Cardinality Ratios:

The cardinality ratio for a binary relationship specifies the number of relationship instances that an entity can participate in. The possible cardinality ratios for binary relationship types are:

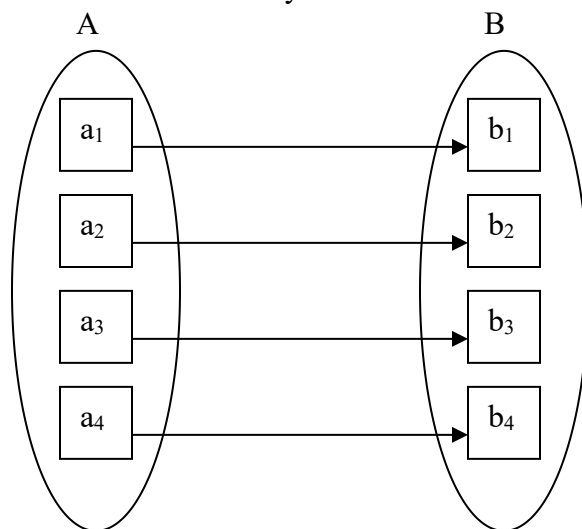
1:1 (One to One),

1:N (One to Many)/N: 1 (Many to One),

M: N (Many to Many).

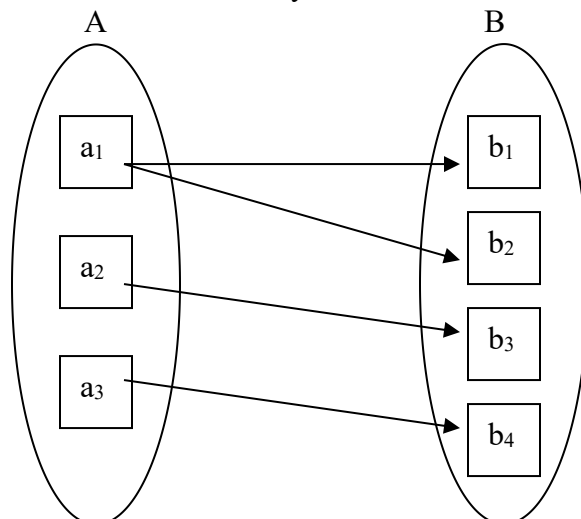
For a binary relationship set r between entity set A and B the mapping cardinality must be one of the following.

One to One: - An entity in A is associated with at most one entity in B and an entity in B is associated with at most one entity in A .



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 constraints that an employee can manage only one department and that a department has only one manager.

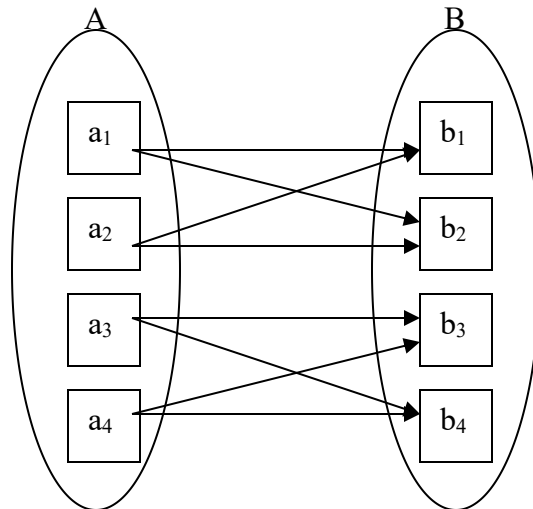
One to Many: - An entity in A is associated with any number of entities in B . An entity in B can be associated with at most one entity in A .



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 numerous employees, but an employee can be related to (work for) only one department.

Many to One:- An entity in A can be associated with at most one entity in B. an entity in B can be associated with any number of entities in A.

Many to Many: - An entity in A can be associated with any number of entities in B and an entity in B is associated with any number of entities in A.



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

Cardinality ratios for binary relationships are displayed on ER diagrams by displaying 1, M, and N on the diamonds as shown in ERD

Participation Constraints /Existence Dependencies

The **participation constraint** specifies whether the existence of an entity depends on its being related to another entity via the relationship type.

There are two types of participation constraints—total and partial—which we illustrate by example. If a company policy states that *every* employee must work for a department, then an employee entity can exist only if it participates in a WORKS_FOR relationship instance. Thus, the participation of EMPLOYEE in WORKS_FOR is called **total participation**, meaning that every entity in "the total set" of employee entities must be related to a department entity via WORKS_FOR. Total participation is also called **existence dependency**.

But we do not expect every employee to manage a department, so the participation of EMPLOYEE in the MANAGES relationship type is **partial**, meaning that *some* or "part of the set of" employee entities are related to a department entity via MANAGES, but not necessarily all. We will refer to the cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type.

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

Attributes of Relationship Types

Relationship types can also have attributes, similar to those of entity types. For example, to record the number of hours per week that an employee works on a particular project, we can include an attribute Hours for the WORKS_ON relationship type. Another example is to include the date on which a manager started managing a department via an attribute StartDate for the MANAGES relationship type.

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

For a 1:N relationship type, a relationship attribute can be migrated *only* to the entity type at the N-side of the relationship. For example, in Figure ERD, if the WORKS_FOR relationship also has an attribute StartDate that indicates when an employee started working for a department, this attribute can be included as an attribute of EMPLOYEE. This is because each employee entity participates in at most one relationship instance in WORKS_FOR.

In both 1:1 and 1:N relationship types, the decision as to where a relationship attribute should be placed—as a relationship type attribute or as an attribute of a participating entity type—is determined subjectively by the schema designer.

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

Weak Entity Types:

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

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

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

DEPENDENT is the partial key. In the worst case, a composite attribute of *all the weak entity's attributes* will be the partial key.

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

In general, any number of levels of weak entity types can be defined; an owner entity type may itself be a weak entity type. In addition, a weak entity type may have more than one identifying entity type and an identifying relationship type of degree higher than two, as we shall illustrate in


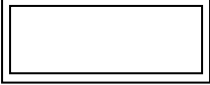





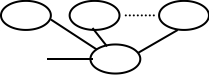
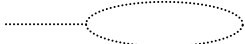
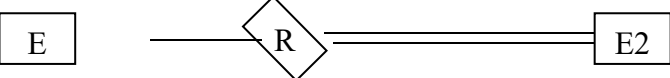
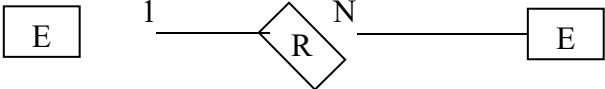
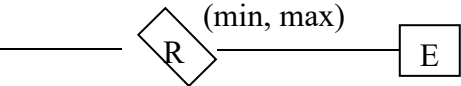
Refining the ER Design for the COMPANY Database

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 listed. In our example, we specify the following relationship types:

1. MANAGES, a 1:1 relationship type between EMPLOYEE and DEPARTMENT. EMPLOYEE participation is partial. DEPARTMENT participation is not clear from the requirements. We question the users, who say that a department must have a manager at all times, which implies total participation. The attribute StartDate is assigned to this relationship type.
2. WORKS_FOR, a 1:N relationship type between DEPARTMENT and EMPLOYEE. Both participations are total.
3. CONTROLS, a 1:N relationship type between DEPARTMENT and PROJECT. The participation of PROJECT is total, whereas that of DEPARTMENT is determined to be partial, after consultation with the users.
4. SUPERVISION, a 1:N relationship type between EMPLOYEE (in the supervisor role) and EMPLOYEE (in the supervisee role). Both participations are determined to be partial, after the users indicate that not every employee is a supervisor and not every employee has a supervisor.
5. WORKS_ON, determined to be an M:N relationship type with attribute Hours, after the users indicate that a project can have several employees working on it. Both participations are determined to be total.
6. DEPENDENTS_OF, a 1:N relationship type between EMPLOYEE and DEPENDENT, which is also the identifying relationship for the weak entity type DEPENDENT. The participation of EMPLOYEE is partial, whereas that of DEPENDENT is total.

After specifying the above six relationship types, we remove from the entity types all attributes that have been refined into relationships. These include Manager and ManagerStartDate from DEPARTMENT; ControllingDepartment from PROJECT; Department, Supervisor, and WorksOn from EMPLOYEE; and Employee from DEPENDENT. It is important to have the least possible redundancy when we design the conceptual schema of a database. If some redundancy is desired at the storage level or at the user view level, it can be introduced later.

Summary of Notation for ER Diagrams

<u>Symbol</u>	<u>Meaning</u>
	ENTITY TYPE
	WEAK ENTITY TYPE
	RELATIONSHIP TYPE
	IDENTIFYING RELATIONSHIP TYPE
	ATTRIBUTE
	KEY ATTRIBUTE
	MULTIVALUED ATTRIBUTE
	COMPOSITE ATTRIBUTE
	DERIVED ATTRIBUTE
	TOTAL PARTICIPATION OF E ₂ IN R
	CARDINALITY RATIO 1:N FOR E ₁ :E ₂ IN R
	STRUCTURAL CONSTRAINT (min, max) ON PARTICIPATION OF E IN R

Enhanced-ER (EER) Model Concepts

Includes all modeling concepts of basic ER. Additional concepts: subclasses/superclasses, specialization/generalization, categories, attribute inheritance

The resulting model is called the enhanced-ER or Extended ER (E2R or EER) model. It is used to model applications more completely and accurately if needed. It includes some object-oriented concepts, such as inheritance .

Subclasses and Superclasses.

An entity type may have additional meaningful subgroupings of its entities

Example: EMPLOYEE may be further grouped into SECRETARY, ENGINEER, MANAGER, TECHNICIAN, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE,...

Each of these groupings is a subset of EMPLOYEE entities. Each is called a subclass of EMPLOYEE. EMPLOYEE is the superclass for each of these subclasses. These are called superclass/subclass relationships.

Example: EMPLOYEE/SECRETARY, EMPLOYEE/TECHNICIAN

These are also called IS-A relationships (SECRETARY IS-A EMPLOYEE, TECHNICIAN IS-A EMPLOYEE, ...).

Note: An entity that is member of a subclass represents the same real-world entity as some member of the superclass. The Subclass member is the same entity in a distinct specific role. An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of the superclass. A member of the superclass can be optionally included as a member of any number of its subclasses

Example: A salaried employee who is also an engineer belongs to the two subclasses ENGINEER and SALARIED_EMPLOYEE

It is not necessary that every entity in a superclass be a member of some subclass

Attribute Inheritance in Superclass / Subclass Relationships

An entity that is member of a subclass inherits all attributes of the entity as a member of the superclass. It also inherits all relationships.

Specialization

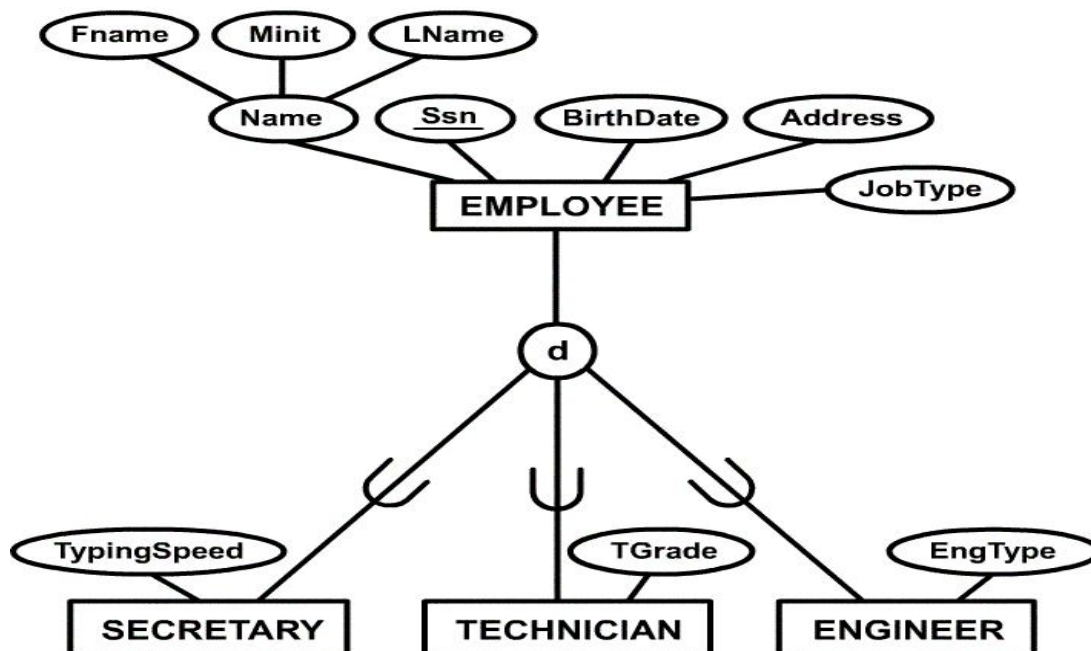
Is the process of defining a set of subclasses of a superclass, The set of subclasses is based upon some distinguishing characteristics of the entities in the superclass.

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

Example: Another specialization of EMPLOYEE based in method of pay is {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}.

Superclass/subclass relationships and specialization can be diagrammatically represented in EER diagrams. Attributes of a subclass are called specific attributes. For example, Typing Speed of SECRETARY. The subclass can participate in specific relationship types. For example, BELONGS_TO of HOURLY_EMPLOYEE

Example of a Specialization



Generalization

It 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

Generalization and Specialization

Diagrammatic notation 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.

We do not use this notation because it is often subjective as to which process is more appropriate for a particular situation. We advocate not drawing any arrows in these situations.

Data Modeling with Specialization and Generalization

A superclass or subclass represents a set of entities. Shown in rectangles in EER diagrams (as are entity types) . Sometimes, all entity sets are simply called classes, whether they are entity types, superclasses, or subclasses

References:

1. Fundamentals of Database Systems: Ramez Elmasri, Shamkant B. Navathe, Pearson.
2. Database System Concepts: Avi Silberschatz · Henry F. Korth · S. Sudarshan, McGraw Hill