Topics to be covered:

- ER Model
- Basic Components of ER Model
- Entity Types and Entity Sets
- Domain of Attributes
- Types of Attributes
- Key attribute
- NULL Values
- Degree of a Relationship
- Constraint on Relationship
- Weak Entities

Learning Outcomes:

Students should be able to:

- Define the following terms: entities, attributes, domain, composite primary key, simple attribute, composite attribute, single-valued attributes, multi-valued attributes, and derived attributes.
- Identify and provide suitable name that is descriptive of the relationship.
- Differentiate between weak and strong relationships.
- Describe and apply constraints on relationship
- Mater sound design principles for logical design of databases using the E-R method
- Design a database schema for a given problem-domain
  - Develop an Entity Relationship Model with the appropriate entities, attributes, relationships, connectivity, and cardinality using Chen notation to represent 1-1, 1-M, and M-N relationships.
- To design ER diagram for any functional requirements

Entity-Relationship (ER) Model

Our focus now is on the second phase, **conceptual design**, for which The **Entity-Relationship (ER) Model** (introduced by Chen in 1976) is a popular high-level conceptual data model. In the ER model, the main concepts are **entity**, **attribute**, and **relationship**.

Entities and Attributes

Entity: An entity represents some "thing" (in the miniworld) that is of interest to us, i.e., about which we want to maintain some data. An entity could represent a physical object (e.g., house, person, automobile, widget) or a less tangible concept (e.g., company, job, academic course, business transaction).

Attribute: An entity is described by its attributes, which are properties characterizing it. Each attribute has a **value** drawn from some **domain** (set of meaningful values). Example: A *PERSON* entity might be described by *Name*, *BirthDate*, *Sex*, etc., attributes, each having a particular value.

What distinguishes an entity from an attribute is that the latter is strictly for the purpose of describing the former and is not, in and of itself, of interest to us. It is sometimes said that an entity has an independent

existence, whereas an attribute does not. In performing data modeling, however, it is not always clear whether a particular concept deserves to be classified as an entity or "only" as an attribute.

We can classify attributes along these dimensions:

- simple/atomic vs. composite
- single-valued vs. multi-valued (or set-valued)
- stored vs. derived (*Note from instructor:* this seems like an implementational detail that ought not be considered at this (high) level of abstraction.)

A **composite** attribute is one that is *composed* of smaller parts. An **atomic** attribute is indivisible or indecomposable.

- **Example 1**: A *BirthDate* attribute can be viewed as being composed of (sub-)attributes *month*, *day*, and *year* (each of which would probably be viewed as being atomic).
- **Example 2**: An *Address* attribute can be viewed as being composed of (sub-)attributes for street address, city, state, and zip code. A street address can itself be viewed as being composed of a number, street name, and apartment number. As this suggests, composition can extend to a depth of two (as here) or more.

To describe the structure of a composite attribute, one can draw a tree. In case we are limited to using text, it is customary to write its name followed by a parenthesized list of its sub-attributes. For the examples mentioned above, we would write
BirthDate(Month, Day, Year)
Address(StreetAddr(StrNum, StrName, AptNum), City, State, Zip)

Single- vs. multi-valued attribute: Consider a *PERSON* entity. The person it represents has (one) *SSN*, (one) *date of birth*, (one, although composite) *name*, etc. But that person may have zero or more academic degrees, dependents, or (if the person is a male living in Utah) spouses! How can we model this via attributes *AcademicDegrees*, *Dependents*, and *Spouses*? One way is to allow such attributes to be *multi-valued* (perhaps *set-valued* is a better term), which is to say that we assign to them a (possibly empty) *set* of values rather than a single value.
To distinguish a multi-valued attribute from a single-valued one, it is customary to enclose the former within curly braces (which makes sense, as such an attribute has a value that is a set, and curly braces are traditionally used to denote sets). Using the *PERSON* example from above, we would depict its structure in text as
PERSON(SSN, Name, BirthDate(Month, Day, Year), { AcademicDegrees(School, Level, Year) }, { Dependents }, ...)
Here we have taken the liberty to assume that each academic degree is described by a school, level (e.g., B.S., Ph.D.), and year. Thus, *AcademicDegrees* is not only multi-valued but also composite. We refer to an attribute that involves some combination of multi-valuedness *and* compositeness as a **complex** attribute.
A more complicated example of a complex attribute is *AddressPhone* in Figure 7.5 (page 207). This attribute is for recording data regarding addresses and phone numbers of a business. The structure of this attribute allows for the business to have several offices, each described by an address and a set of phone numbers that ring into that office. Its structure is given by
{ AddressPhone({ Phone(AreaCode, Number) }, Address(StrAddr(StrNum, StrName, AptNum), City, State, Zip)) }

Stored vs. derived attribute: Perhaps *independent* and *derivable* would be better terms for these (or *non-redundant* and *redundant*). In any case, a *derived* attribute is one whose value can be calculated from the values of other attributes, and hence need not be stored. **Examples:** *Age* can be calculated from *BirthDate*, assuming that the current date is accessible. *GPA* can be calculated, assuming that the necessary data regarding courses and grades is accessible.

The Null value: In some cases a particular entity might not have an applicable value for a particular attribute. Or that value may be unknown.

Example: The attribute *DateOfDeath* is not applicable to a living person and its correct value may be unknown for some persons who have died.
In such cases, we use a special attribute value (non-value?), called **null**. There has been some argument among database experts about whether a different approach (such as having distinct values for *not applicable* and *unknown*) would be superior.

Entity Types, Entity Sets, Keys, and Domains (Value Sets)
Above we mentioned the concept of a *PERSON* entity, i.e., a representation of a particular person via the use of attributes such as *Name*, *Sex*, etc. As a general rule, for each entity type there will be multiple (perhaps even thousands or millions of) entities of that type about which we want to store data in the database, each of them described by the same collection of attributes. Of course, the *values* of those attributes will differ from one entity to another (e.g., one person will have the name "Mary" and another will have the name "Rumpelstiltskin"). Just as likely is that we will want our database to store information about other kinds of entities, such as business transactions or academic courses, which will be described by entirely different collections of attributes.

This illustrates the distinction between entity types and entity instances. An **entity type** serves as a template for a collection of **entity instances**, all of which are described by the same collection of attributes. That is, an entity type is analogous to a **class** in object-oriented programming and an entity instance is analogous to a particular object (i.e., instance of a class). (Or, even simpler, an entity type and instance, respectively, are analogous to a data type (e.g., integer) and a value of that type (e.g., 57).)
In ER modeling, we deal only with entity types, not with instances. In an ER diagram, each entity type is denoted by a rectangular box.

An **entity set** is the collection of all entities of a particular type that exist, in a database, at some moment in time.

Key Attributes of an Entity Type: A minimal collection of attributes (often only one) that, by design, distinguishes any two (simultaneously-existing) entities of that type. In other words, if attributes A_1 through A_m together form a key of entity type *E*, and *e* and *f* are two entities of type *E* existing at the same time, then, in at least one of the attributes A_i ($0 < i <= m$), *e* and *f* must have distinct values.
An entity type could have more than one key. (An example of the `CAR` entity type is postulated to have both { *Registration(RegistrationNum, State)* } and { *VehicleID* } as keys.)

Domains (Value Sets) of Attributes: The domain of an attribute is the "universe of values" from which its value can be drawn. In other words, an attribute's domain specifies its set of allowable values. The concept is similar to **data type**.

Relationship Types, Sets, Roles, and Structural Constraints
Having presented a preliminary database schema for COMPANY, it is now convenient to clarify the concept of a **relationship** (which is the last of the three main concepts involved in the ER model).

Relationship: This is an association between two entities. As an example, one can imagine a `STUDENT` entity being associated to an `ACADEMIC_COURSE` entity via, say, an `ENROLLED_IN` relationship.
Whenever an attribute of one entity type refers to an entity (of the same or of a different entity type), we say that a relationship exists between the two entity types.

From our preliminary COMPANY schema, we identify the following **relationship types** (using descriptive names and ordering the participating entity types so that the resulting phrase will be in active voice rather than passive):

- EMPLOYEE MANAGES DEPARTMENT (arising from *Manager* attribute in DEPARTMENT)
- DEPARTMENT CONTROLS PROJECT (arising from *ControllingDept* attribute in PROJECT and the *Projects* attribute in DEPARTMENT)
- EMPLOYEE WORKS_FOR DEPARTMENT (arising from *Dept* attribute in EMPLOYEE and the *Employees* attribute in DEPARTMENT)
- EMPLOYEE SUPERVISES EMPLOYEE (arising from *Supervisor* attribute in EMPLOYEE)
- EMPLOYEE WORKS_ON PROJECT (arising from *WorksOn* attribute in EMPLOYEE and the *Workers* attribute in PROJECT)
- DEPENDENT DEPENDS_ON EMPLOYEE (arising from *Employee* attribute in DEPENDENT and the *Dependents* attribute in EMPLOYEE)

In ER diagrams, relationship types are drawn as diamond-shaped boxes connected by lines to the entity types involved. Note that attributes are depicted by ovals connected by lines to the entity types they describe (with multi-valued attributes in double ovals and composite attributes depicted by trees). The original attributes that gave rise to the relationship types are absent, having been replaced by the relationship types.

A **relationship set** is a set of instances of a relationship type. If, say, `R` is a relationship type that relates entity types `A` and `B`, then, at any moment in time, the relationship set of `R` will be a set of ordered pairs *(x,y)*, where *x* is an instance of `A` and *y* is an instance of `B`. What this means is that, for example, if our COMPANY miniworld is, at some moment, such that employees e_1, e_3, and e_6 work for department d_1, employees e_2 and e_4 work for department d_2, and employees e_5 and e_7 work for department d_3, then the `WORKS_FOR` **relationship set** will include as **instances** the ordered pairs (e_1, d_1), (e_2, d_2), (e_3, d_1), (e_4, d_2), (e_5, d_3), (e_6, d_1), and (e_7, d_3).

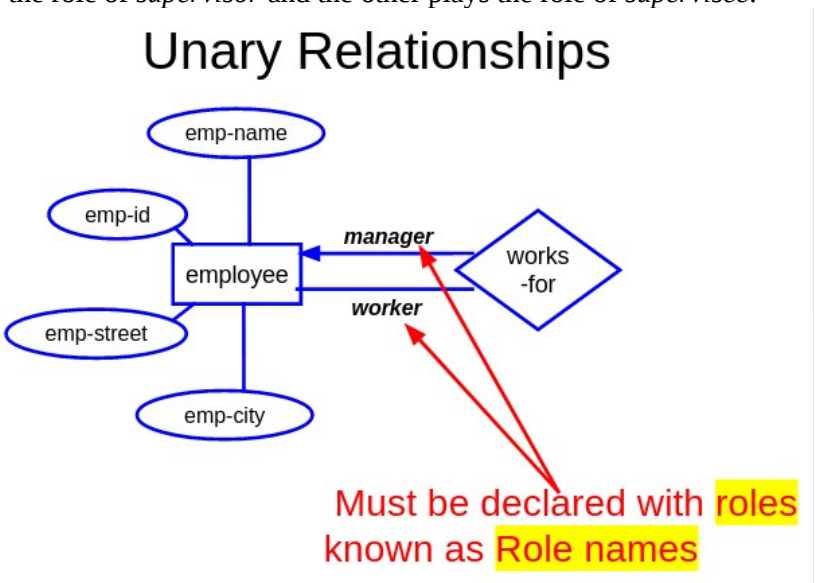
Ordering of entity types in relationship types: Note that the order in which we list the entity types in describing a relationship is of little consequence, except that the relationship name (for purposes of clarity) ought to be consistent with it. For example, if we swap the two entity types in each of the first two relationships listed above, we should rename them `IS_MANAGED_BY` and `IS_CONTROLLED_BY`, respectively.

Degree of a relationship type: Also note that, in our COMPANY example, all relationship instances will be ordered pairs, as each relationship associates an instance from one entity type with an instance of another (or the same, in the case of `SUPERVISES`) entity type. Such relationships are said to be *binary*, or to have *degree* two. Relationships with degree three (called *ternary*) or more are also possible, but they do not arise as often in practice. a relationship `SUPPLY` (perhaps not the best choice for a name) has as instances ordered triples of suppliers, parts, and projects, with the intent being that inclusion of the ordered triple *(s_2, p_4, j_1)*, for example, indicates that supplier s_2 supplied part p_4 to project j_1.

Roles in relationships: Each entity that participates in a relationship plays a particular *role* in that relationship, and it is often convenient to refer to that role using an appropriate name. For example, in each instance of a `WORKS_FOR` relationship set, the employee entity plays the role of *worker* or (surprise!) *employee* and each department plays the role of *employer* or (surprise!) *department*. Indeed, as this example suggests, often it is best to use the same name for the role as for the corresponding entity type.

An exception to this rule occurs when the same entity type plays two (or more) roles in the same relationship. (Such relationships are said to be *recursive*, which I find to be a misleading use of that term.

A better term might be *self-referential*.) For example, in each instance of a `SUPERVISES` relationship set, one employee plays the role of *supervisor* and the other plays the role of *supervisee*.

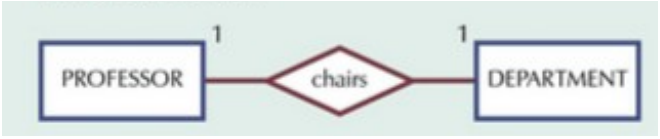


Constraints on Binary Relationship Types

Often, in order to make a relationship type be an accurate model of the miniworld concepts that it is intended to represent, we impose certain constraints that limit the possible corresponding relationship sets. (That is, a constraint may make "invalid" a particular set of instances for a relationship type.)

There are two main kinds of relationship constraints (on binary relationships). For illustration, let *R* be a relationship set consisting of ordered pairs of instances of entity types *A* and *B*, respectively.

- **cardinality ratio**:
  - **1:1 (one-to-one)**: Under this constraint, no instance of *A* may particpate in more than one instance of *R*; similarly for instances of *B*. In other words, if (a_1, b_1) and (a_2, b_2) are (distinct) instances of *R*, then neither $a_1 = a_2$ nor $b_1 = b_2$.
    Example: Our informal description of COMPANY says that every department has one employee who manages it. If we also stipulate that an employee may not (simultaneously) play the role of manager for more than one department, it follows that MANAGES is 1:1.



  - **1:N (one-to-many)**: Under this constraint, no instance of *B* may participate in more than one instance of *R*, but instances of *A* are under no such restriction. In other words, if (a_1, b_1) and (a_2, b_2) are (distinct) instances of *R*, then it cannot be the case that $b_1 = b_2$.
    Example: CONTROLS is 1:N because no project may be controlled by more than one department. On the other hand, a department may control any number of projects, so there is no restriction on the number of relationship instances in which a particular department instance may participate. For similar reasons, SUPERVISES is also 1:N.
  - **N:1 (many-to-one)**: This is just the same as 1:N but with roles of the two entity types reversed.
    Example: WORKS_FOR and DEPENDS_ON are N:1.

- **M:N (many-to-many)**: Under this constraint, there are no restrictions. (Hence, the term applies to the absence of a constraint!)
  Example: WORKS_ON is M:N, because an employee may work on any number of projects and a project may have any number of employees who work on it.

Suppose that, in designing a database, we decide to include a binary relationship *R*, as described above, that relates entity types *A* and *B*, respectively. To determine how *R* should be constrained, with respect to cardinality ratio, the questions you should ask are these:
May a given entity of type B be related to multiple entities of type A?
May a given entity of type A be related to multiple entities of type B?
The pair of answers you get maps into the four possible cardinality ratios as follows:
(yes, yes) --> M:N
(yes, no) --> N:1
(no, yes) --> 1:N
(no, no) --> 1:1

- **participation**: specifies whether or not the existence of an entity depends upon its being related to another entity via the relationship.
  - **total participation (or existence dependency)**: To say that entity type *A* is constrained to **participate totally** in relationship *R* is to say that if (at some moment in time) *R*'s instance set is
    $\{ (a_1, b_1), (a_2, b_2), \ldots (a_m, b_m) \}$,

    then (at that same moment) *A*'s instance set must be $\{ a_1, a_2, \ldots, a_m \}$. In other words, there can be no member of *A*'s instance set that does not participate in at least one instance of *R*.
    According to our informal description of COMPANY, every employee must be assigned to some department. That is, every employee instance must participate in at least one instance of `WORKS_FOR`, which is to say that *EMPLOYEE* satisfies the total participation constraint with respect to the `WORKS_FOR` relationship.
    In an ER diagram, if entity type *A* must participate totally in relationship type `R`, the two are connected by a double line.
    partial participation: the absence of the total participation constraint! (E.g., not every employee has to participate in `MANAGES`; hence we say that, with respect to `MANAGES`, *EMPLOYEE* participates partially. This is not to say that for all employees to be managers is not allowed; it only says that it need not be the case that all employees are managers.

Attributes of Relationship Types
Relationship types, like entity types, can have attributes. A good example is `WORKS_ON`, each instance of which identifies an employee and a project on which (s)he works. In order to record (as the specifications indicate) how many hours are worked by each employee on each project, we include *Hours* as an attribute of `WORKS_ON`. In the case of an M:N relationship type (such as `WORKS_ON`), allowing attributes is vital. In the case of an N:1, 1:N, or 1:1 relationship type, any attributes can be assigned to the entity type opposite from the 1 side. For example, the *StartDate* attribute of the `MANAGES` relationship type can be given to either the *EMPLOYEE* or the *DEPARTMENT* entity type.

Weak Entity Types: An entity type that has no set of attributes that qualify as a key is called **weak**. (Ones that do are **strong**.)
An entity of a weak identity type is uniquely identified by the specific entity to which it is related (by a so-called **identifying relationship** that relates the weak entity type with its so-called **identifying** or **owner entity type**) in combination with some set of its own attributes (called a *partial key*).
Example: A *DEPENDENT* entity is identified by its first name together with the *EMPLOYEE* entity to which it is related via `DEPENDS_ON`. (Note that this wouldn't work for former heavyweight boxing champion George Foreman's sons, as they all have the name "George"!)
Because an entity of a weak entity type cannot be identified otherwise, that type has a **total participation constraint** (i.e., **existence dependency**) with respect to the identifying relationship.
This should not be taken to mean that any entity type on which a total participation constraint exists is weak. For example, DEPARTMENT has a total participation constraint with respect to `MANAGES`, but it is not weak.
In an ER diagram, a weak entity type is depicted with a double rectangle and an identifying relationship type is depicted with a double diamond.

Design Choices for ER Conceptual Design: Sometimes it is not clear whether a particular miniworld concept ought to be modeled as an entity type, an attribute, or a relationship type. Here are some guidelines (given with the understanding that schema design is an iterative process in which an initial design is refined repeatedly until a satisfactory result is achieved):

- As happened in our development of the ER model for COMPANY, if an attribute of entity type *A* serves as a reference to an entity of type *B*, it may be wise to refine that attribute into a binary relationship involving entity types *A* and *B*. It may well be that *B* has a corresponding attribute referring back to *A*, in which case it, too, is refined into the aforementioned relationship. In our COMPANY example, this was exemplified by the *Projects* and *ControllingDept* attributes of *DEPARTMENT* and *PROJECT*, respectively.
- An attribute that exists in several entity types may be refined into its own entity type. For example, suppose that in a UNIVERSITY database we have entity types *STUDENT*, *INSTRUCTOR*, and *COURSE*, all of which have a *Department* attribute. Then it may be wise to introduce a new entity type, *DEPARTMENT*, and then to follow the preceding guideline by introducing a binary relationship between *DEPARTMENT* and each of the three aforementioned entity types.
- An entity type that is involved in very few relationships (say, zero, one, or possibly two) could be refined into an attribute (of each entity type to which it is related).

Solved 2 Problems (in class):

Q.1 > Draw an ER diagram for the following application from the manufacturing industry:

1. Each supplier has a unique name.
2. More than one supplier can be located in the same city.
3. Each part has a unique part number.
4. Each part has a colour.
5. A supplier can supply more than one part.
6. A part can be supplied by more than one supplier.
7. A supplier can supply a fixed quantity of each part.

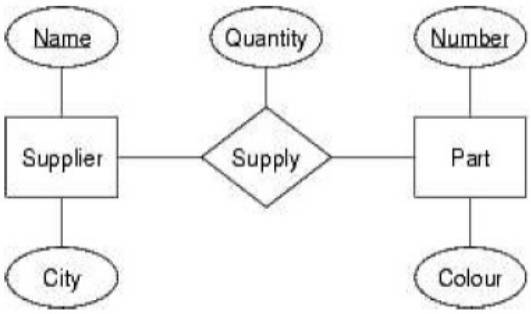


Figure 1: Solution For Q.1

Q.2 > Construct an E-R diagram for a car-insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents.

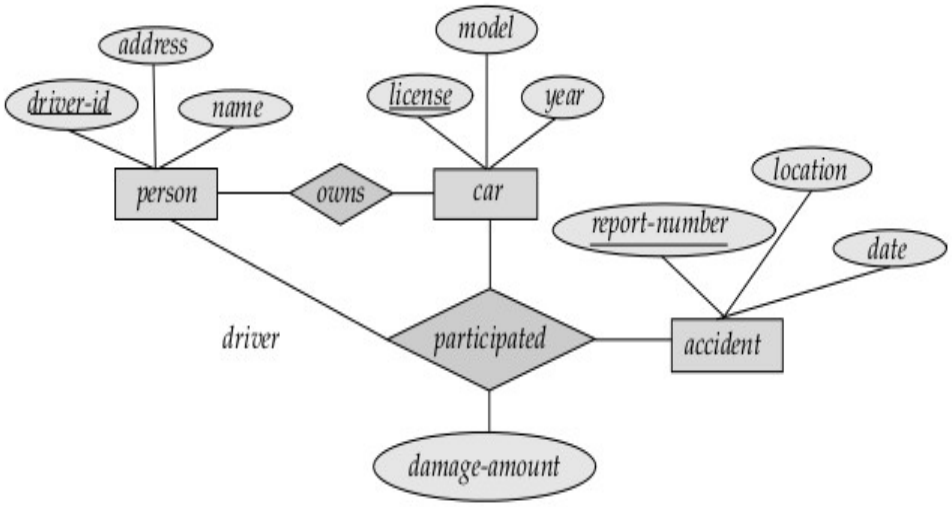


Figure 2: Solution For Q.2

Source:

1. Ramez Elmasri, Shamkant B. Navathe
   Fundamentals of Database System
   Fourth Edition

Long Answer Type Questions:

1. What are the basic components of ER model.
2. What do you mean by an attributes. Explain various types of attributes.
3. Write a short note on:
   a. Weak Entity Type
   b. Role Name
   c. Constraint on relationship
   d. Constraint of Generalization and Specialization
   e. Attributes of Relationship Types
   f. Domain of attributes

4. Construct an E-R diagram for a car-insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents.

5. Define the following term with suitable example :-
   (i) Weak and strong entity set
   (ii) Mapping cardinalities