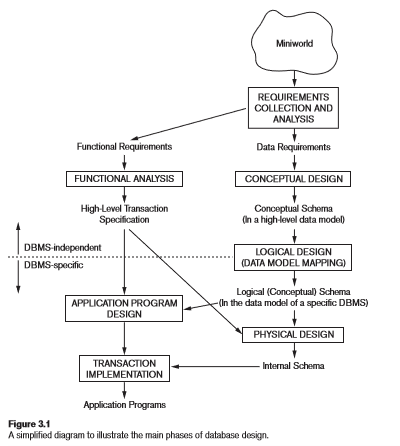
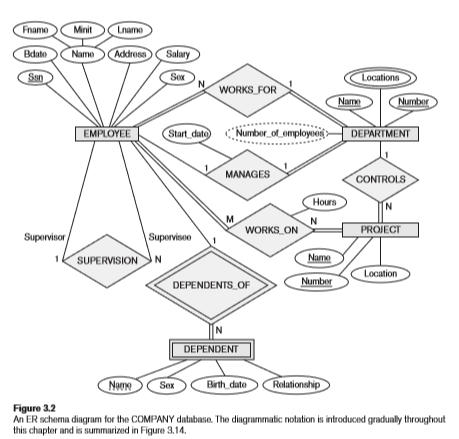
**CHAPTER 2**

**2.1Using High-Level Conceptual Data Models for Database Design**



* Figure 2.1 shows a simplified overview of the database design process. The first step shown is requirements collection and analysis. During this step, the database designers interview prospective database users to understand and document their data requirements.
* 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.
* During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user queries and operations identified during functional analysis.
* The next step in database design is the actual implementation of the database, using a commercial DBMS. 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.

**2.2 A sample Database Application**



**2.3 Entity Types, Entity Sets, Attributes, and Keys**

Our focus now is on the second phase, **conceptual design**, for which The **Entity Relationship (ER) Model** 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). **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.

**Types of Attributes**

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 for month, day, and year.
* **Example 2**: An *Address* attribute (Figure 1.14, page 64) 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. 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.

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. **Example:***Age* can be calculated from *BirthDate*, assuming that the current date 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. Or, in the case of a multi-valued attribute, the appropriate value might be the empty set.

*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 in the database literature about whether a different approach (such as having distinct values for *not applicable* and *unknown*) would be superior.

## Entity Types, Entity Sets, Keys, and Value sets

An **entity type** defines a collection (or set) of entities that have the same attributes. Each entity type in the database is described by its name and attributes.

An **entity set (entity collection)** 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**: 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. (For example: adhaar number)

# 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.

# In Figure, 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 COMPANY database

Using the above structured description as a guide, we get the following preliminary design for entity types and their attributes in the COMPANY database:

* DEPARTMENT(Name, Number, { Locations }, Manager, ManagerStartDate, {

Employees }, { Projects })

* PROJECT(Name, Number, Location, { Workers }, ControllingDept)
* EMPLOYEE(Name(FName, MInit, LName), SSN, Sex, Address, Salary, BirthDate,

Dept, Supervisor, { Dependents }, { WorksOn(Project, Hours) })

* DEPENDENT(Employee, FirstName, Sex, BirthDate, Relationship)

## 2.4 Relationship Types, Sets, Roles, and Structural Constraints

**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.

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).

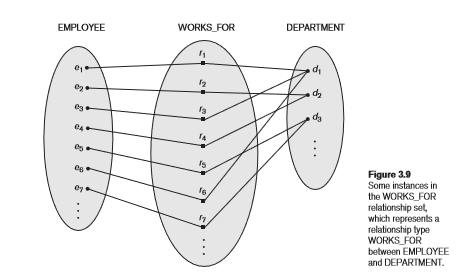
A **relationship type** R among n entity types E1, E2, . . . , En defines a set of associations—or a relationship set—among entities from these entity types. Similar to the case of entity types and entity sets, a relationship type and its corresponding relationship set are customarily referred to by the same name, R.

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

Hence, a relationship set is a mathematical relation on E1, E2, . . . , En; a lternatively, it can be defined as a subset of the Cartesian product of the entity sets E1 × E2 × . . . × En.

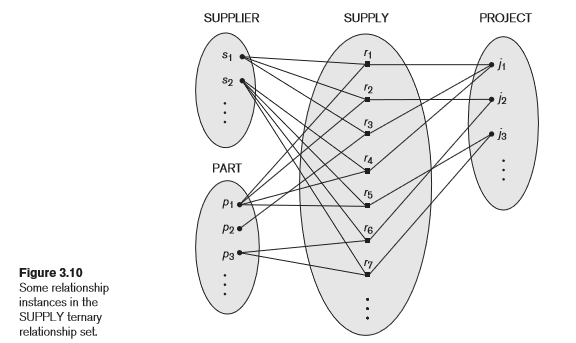
Each of the entity types E1, E2, . . . , En is said to participate in the relationship type R; similarly, each of the individual entities e1, e2, . . . , en is said to participate in the relationship instance ri = (e1, e2, . . . , en).

Each relationship instance ri in R is an association of entities, where the association includes exactly one entity from each participating entity type.



EMPLOYEE and DEPARTMENT, which associates each employee with the department for which the employee works. Each relationship instance in the relationship set WORKS\_FOR associates one EMPLOYEE entity and one DEPARTMENT entity. Figure 3.9 illustrates this example, where each relationship instance ri is shown connected to the EMPLOYEE and DEPARTMENT entities that participate inri. In the miniworld represented by Figure 3.9, the employees e1, e3, and e6 work for department d1; the employees e2 and e4 work for department d2; and the employees e5 and e7 work for department d3.

**Degree of a Relationship Type.** The degree of a relationship type is the number of participating entity types. Hence, the WORKS\_FOR relationship is of degree two. A relationship type of degree two is called binary, and one of degree three is called ternary.



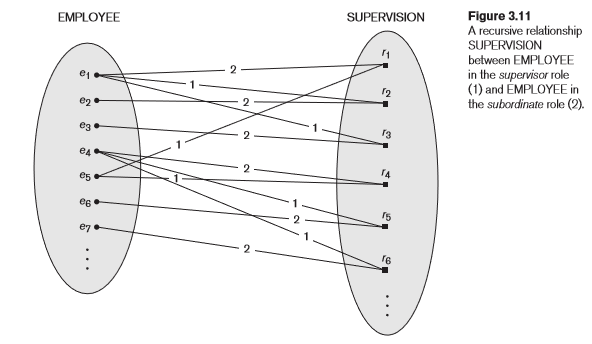
## Relationships as Attributes. It is sometimes convenient to think of a binary relationship type in terms of attributes, as we discussed in Section 3.3.3. Consider the WORKS\_FOR relationship type in Figure 3.9. One can think of an attribute called Department of the EMPLOYEE entity type, where the value of Department for each EMPLOYEE entity is (a reference to) the DEPARTMENT entity for which that employee works. Hence, the value set for this Department attribute is the set of all DEPARTMENT entities, which is the DEPARTMENT entity set.

**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 it 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.

## Recursive relationships or self-referencing relationships

If the same entity type participates more than once in a relationship type in different roles.

Example: The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity set. Hence, the EMPLOYEE entity type participates twice in SUPERVISION: once in the role of supervisor (or boss), and once in the role of supervisee (or subordinate).



## Constraints on Binary Relationship Types

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 *(a1, b1)* and *(a2, b2)* are (distinct) instances of *R*, then neither *a1 = a2* nor *b1 = b2*. **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 *(a1, b1)* and *(a2, b2)* are (distinct) instances of *R*, then it cannot be the case that *b1 = b2*.

**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.

Notice the notation in Figure 3.2 for indicating each relationship type's cardinality ratio. Suppose that, in designing a database, we decide to include a binary relationship *R* as described above (which 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:

**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

*{ (a1, b1), (a2, b2), ... (am, bm) }*, then (at that same moment) *A*'s instance set must be *{ a1, a2, ..., am }*. 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. See Figure 3.2.

* **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. (See Figure 3.2 again.) 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.

**3.5 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).

