UNIT-2

Relational Model Concepts

Domain: A (usually named) set/universe of *atomic* values, where by "atomic" we mean simply that, from the point of view of the database, each value in the domain is indivisible (i.e., cannot be broken down into component parts).

Examples of domains:

USA_phone_number: string of digits of length ten
SSN: string of digits of length nine
Name: string of characters beginning with an upper case
letter ○ GPA: a real number between 0.0 and 4.0
Sex: a member of the set { female, male }
Dept_Code: a member of the set { CMPS, MATH, ENGL, PHYS, PSYC, ... }

These are all *logical* descriptions of domains. For implementation purposes, it is necessary to provide descriptions of domains in terms of concrete **data types** (or **formats**) that are provided by the DBMS (such as String, int, boolean), in a manner analogous to how programming languages have intrinsic data types.

Attribute: the *name* of the role played by some value (coming from some domain) in the context of a **relational schema**. The domain of attribute A is denoted dom(A).

Tuple: A tuple is a mapping from attributes to values drawn from the respective domains of those attributes. A tuple is intended to describe some entity (or relationship between entities) in the miniworld.

As an example, a tuple for a PERSON entity might be

{ Name --> "Rumpelstiltskin", Sex --> Male, IQ --> 143 }

Relation: A (named) set of tuples all of the same form (i.e., having the same set of attributes). The term **table** is a loose synonym. (Some database purists would argue
that a table is "only" a physical manifestation of a relation.)

$R(A_1, A_2, ...,$

Relational Schema: used for describing (the structure of) a relation. E.g.,
$A_n)$ says that R is a relation with *attributes* A_1, ... A_n. The **degree** of a relation is the number of attributes it has, here *n*.

Example: STUDENT(Name, SSN, Address)

One would think that a "complete" relational schema would also specify the domain of each attribute.

Relational Database: A collection of **relations**, each one consistent with its specified relational schema.

Characteristics of Relations

Ordering of Tuples: A relation is a *set* of tuples; hence, there is no order associated with them. That is, it makes no sense to refer to, for example, the 5th tuple in a relation. When a relation is depicted as a table, the tuples are necessarily listed in *some* order, of course, but you should attach no significance to that order. Similarly, when tuples are represented on a storage device, they must be organized in *some* fashion, and it may be advantageous, from a performance standpoint, to organize them in a way that depends upon their content.

Ordering of Attributes: A tuple is best viewed as a mapping from its attributes (i.e., the names we give to the roles played by the values comprising the tuple) to the corresponding values. Hence, the order in which the attributes are listed in a table is irrelevant. (Note that, unfortunately, the set theoretic operations in relational algebra (at least how E&N define them) make implicit use of the order of the attributes. Hence, E&N view attributes as being arranged as a sequence rather than a set.)

Values of Attributes: For a relation to be in *First Normal Form*, each of its attribute domains must consist of atomic (neither composite nor multi-valued) values. Much of the theory underlying the relational model was based upon this assumption.

The **Null** value: used for *don't know, not applicable*.

Interpretation of a Relation: Each relation can be viewed as a **predicate** and each tuple in that relation can be viewed as an assertion for which that predicate is satisfied (i.e., has value **true**) for the combination of values in it. In other words, each tuple represents a fact. Example The first tuple listed means: There exists a student having name Benjamin Bayer, having SSN 305-61-2435, having age 19, etc.

Keep in mind that some relations represent facts about entities (e.g., students) whereas others represent facts about relationships (between entities). (e.g., students and course sections).

The **closed world assumption** states that the only true facts about the miniworld are those represented by whatever tuples currently populate the database.

Relational Model Notation:

$R(A_1, A_2, ..., A_n)$ is a relational schema of degree *n* denoting that there is a relation *R* having as its attributes $A_1, A_2, ..., A_n$.

By convention, *Q*, *R*, and *S* denote relation names.

By convention, *q*, *r*, and *s* denote relation states. For example, *r(R)* denotes one possible state of relation *R*. If *R* is understood from context, this could be written, more simply, as *r*.

By convention, *t*, *u*, and *v* denote tuples.

The "dot notation" *R.A* (e.g., STUDENT.Name) is used to qualify an attribute name, usually

for the purpose of distinguishing it from a same-named attribute in a different relation(e.g., DEPARTMENT.Name).

Relational Model Constraints and Relational Database Schemas

Constraints on databases can be categorized as follows:

inherent model-based: Example: no two tuples in a relation can be duplicates (because a relation is a set of tuples)

schema-based: can be expressed using DDL; this kind is the focus of this section.

application-based: are specific to the "business rules" of the miniworld and typically difficult or impossible to express and enforce within the data model. Hence, it is left to application programs to enforce.

Elaborating upon **schema-based constraints**:

Domain Constraints: Each attribute value must be either **null** (which is really a *non-value*) or drawn from the domain of that attribute. Note that some DBMS's allow you to impose the **not null** constraint upon an attribute, which is to say that that attribute may not have the (non-)value **null**.
Key Constraints: A relation is a *set* of tuples, and each tuple's "identity" is given by the values of its attributes. Hence, it makes no sense for two tuples in a relation to be identical (because then the two tuples are actually one and the same tuple). That is, no two tuples may have the same combination of values in their attributes.

Usually the miniworld dictates that there be (proper) subsets of attributes for which no two tuples may have the same combination of values. Such a set of attributes is called a **superkey** of its relation. From the fact that no two tuples can be identical, it follows that the set of all attributes of a relation constitutes a superkey of that relation.

A **key** is a *minimal superkey,* i.e., a superkey such that, if we were to remove any of its attributes, the resulting set of attributes fails to be a superkey.

Example: Suppose that we stipulate that a faculty member is uniquely identified by *Name* and *Address* and also by *Name* and *Department*, but by no single one of the three attributes mentioned. Then *{ Name, Address, Department }* is a (non-minimal) superkey and each of *{ Name, Address }* and *{ Name, Department }* is a key (i.e., minimal superkey).

Candidate key: any key! (Hence, it is not clear what distinguishes a key from a candidate key.)

Primary key: a key chosen to act as the means by which to identify tuples in a relation. Typically, one prefers a primary key to be one having as few attributes as possible.

Relational Databases and Relational Database Schemas

A **relational database schema** is a set of schemas for its relations together with a set of **integrity constraints**.

A **relational database state/instance/snapshot** is a set of states of its relations such that no integrity constraint is violated.

Entity Integrity, Referential Integrity, and Foreign Keys

Entity Integrity Constraint: In a tuple, none of the values of the attributes forming the relation's primary key may have the (non-)value **null**. Or is it that at least one such attribute must have a non-null value? In my opinion, E&N do not make it clear!

Referential Integrity Constraint: A **foreign key** of relation *R* is a set of its attributes intended to be used (by each tuple in *R*) for identifying/referring to a tuple in some relation *S*. (*R* is called the *referencing* relation and *S* the *referenced* relation.) For this to make sense, the set of attributes of *R* forming the foreign key should "correspond to" some superkey of *S*. Indeed, by definition we require this superkey to be the primary key of *S*.

This constraint says that, for every tuple in *R*, the tuple in *S* to which it refers must actually be in *S*. Note that a foreign key may refer to a tuple in the same relation and that a foreign key may be part of a primary key (indeed, for weak entity types, this will always occur). A foreign key may have value **null** (necessarily in all its attributes??), in which case it does not refer to any tuple in the referenced relation.

Semantic Integrity Constraints: application-specific restrictions that are unlikely to be expressible in DDL. Examples:salary of a supervisee cannot be greater than that of her/his supervisor salary of an employee cannot be lowered

Update Operations and Dealing with Constraint Violations

For each of the *update* operations (Insert, Delete, and Update), we consider what kinds of constraint violations may result from applying it and how we might choose to react.

Insert:domain constraint violation: some attribute value is not of correct domain entity integrity violation: key of new tuple is **null** key constraint violation: key of new tuple is same as existing one referential integrity violation: foreign key of new tuple refers to non-existent tuple

Ways of dealing with it: reject the attempt to insert! Or give user opportunity to try againwith different attribute values.

Delete:referential integrity violation: a tuple referring to the deleted one exists. Three options for dealing with it:

Reject the deletion

Attempt to **cascade** (or **propagate**) by deleting any referencing tuples (plus those that reference them, etc., etc.)

modify the foreign key attribute values in referencing tuples to **null** or to some valid value referencing a different tuple

Update:

Key constraint violation: primary key is changed so as to become same as another tuple's referential integrity violation:

- foreign key is changed and new one refers to nonexistent tuple
- primary key is changed and now other tuples that had referred to this one violate the constraint

Keys: There is no duplicate tuples within a relation. So we need to be able to identify one or more attributes called relation keys.

Super key- An attribute or set of attributes that uniquely identifies a tuple within a relation is called super key.

Candidate key- A super key has no proper subset is a super key within a relation is called candidate key. It having 2 properties i.e., Uniqueness, Irreducibility.

Composite key- If a key consists of more than one attributes then it is called as composite key

Primary key- The candidate key that is selected to identify tuples uniquely within the relation is called as primary key.

Alternate keys- The candidate key that are not selected to be the primary key

Foreign key- An attribute or set of attributes, within one relation that matches the candidate key of some relation is called foreign key.

Entity-Relationship (ER) Model

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.

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 to 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 3.4, 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 tree (as in the aforementioned Figure 3.4). 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 3.5 (page 65). 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. **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 Date Of Death 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.

2.8.2: Entity Types, Entity Sets, Keys, and Domains

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. Chances are good that, in a database in which one such entity exists, we will want many others of the same kind to exist also, 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).

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 this appears in Figure 3.7, page 67, in which 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**.

Example Database Application: COMPANY

Suppose that Requirements Collection and Analysis results in the following (informal) description of the COMPANY miniworld:

The company is organized as a collection of **departments**.

- Each department
  - has a unique name
  - has a unique number
  - is associated with a set of locations
  - has a particular employee who acts as its manager (and who assumed that position on some date)
  - has a set of employees assigned to it
  - controls a set of projects
- Each project
  - has a unique name
  - has a unique number
  - has a single location
  - has a set of employees who work on it
  - is controlled by a single department
- Each employee
  - has a name
  - has a SSN that uniquely identifies her/him
  - has an address
  - has a salary
  - has a sex
  - has a birthdate
  - has a direct supervisor
  - has a set of dependents
  - is assigned to one department
  - works some number of hours per week on each of a set of projects (which need not all be controlled by the same department)
- Each dependent
  - has first name
  - has a sex
  - has a birthdate
  - is related to a particular employee in a particular way (e.g., child, spouse, pet)
  - is uniquely identified by the combination of her/his first name and the employee of which (s)he is a dependent

2.8.3 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(<u>Name</u>, <u>Number</u>, { Locations }, Manager, ManagerStartDate, { Employees }, { Projects })
- PROJECT(<u>Name</u>, <u>Number</u>, Location, { Workers }, ControllingDept)
- EMPLOYEE(Name(FName, MInit, LName), <u>SSN</u>, Sex, Address, Salary, BirthDate, Dept, Supervisor, { Dependents }, { WorksOn(Project, Hours) })
- DEPENDENT(<u>Employee, FirstName</u>, Sex, BirthDate, Relationship)

Remarks: Note that the attribute WorksOn of EMPLOYEE (which records on which projects the employee works) is not only multi-valued (because there may be several such projects) but also composite, because we want to record, for each such project, the number of hours per week that the employee works on it. Also, each candidate key has been indicated by underlining.

For similar reasons, the attributes Manager and ManagerStartDate of DEPARTMENT really ought to be combined into a single composite attribute. Not doing so causes little or no harm, however, because these are single-valued attributes. Multi-valued attributes would pose some difficulties, on the other hand. Suppose, for example, that a department could have two or more managers, and that some department had managers Mary and Harry, whose start dates were 10-4-1999 and 1-13-2001, respectively. Then the values of the Manager and ManagerStartDate attributes should be { Mary, Harry } and { 10-4-1999, 1-13-2001 }. But from these two attribute values, there is no way to determine which manager started on which date. On the other hand, by recording this data as a set of ordered pairs, in which each pair identifies a manager and her/his starting date, this deficiency is eliminated. End of Remarks

2.9 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 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. See Figure 3.2, page 62. 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). See Figure 3.9 on page 71 for a graphical depiction of this.

2.9.1 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.

2.9.2 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) relationship type. Such relationships are said to be binary, or to have degree two. Relationships with degree three (called ternary) or more are also possible, although not as common. This is illustrated in Figure 3.10 (page 72), where 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.

2.9.3 Constraints on 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.

    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:
  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), ... (a_m, b_m) }, then (at that same moment) A's instance set must be { a_1, a_2, ..., 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. 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.

2.10 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.

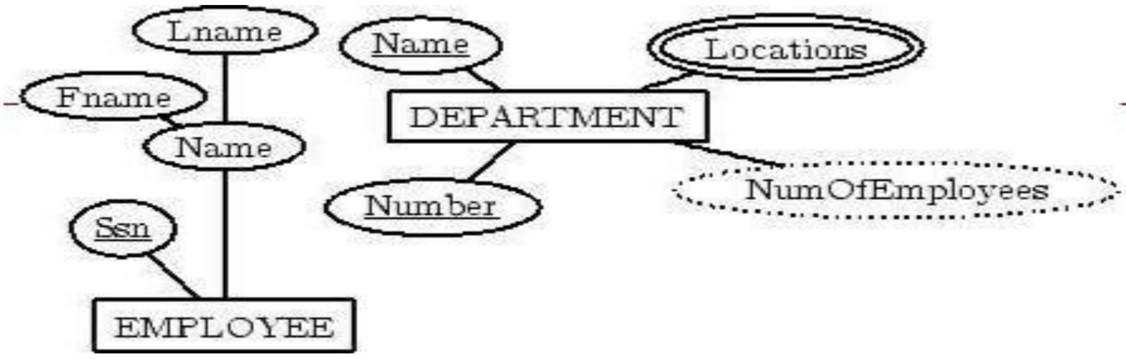
In an ER diagram, a weak entity type is depicted with a double rectangle and an identifying relationship type is depicted with a double diamond.

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).

Relational Database Design Using ER-to-Relational Mapping

Step 1: For each **regular (strong) entity type** E in the ER schema, create a relation R that includes all the simple attributes of E.



EMPLOYEE

SSN	Lname	Fname

DEPARTMENT

NUMBER	NAME

Step 2: For each **weak entity type** W in the ER schema with owner entity type E, create a relation R, and include all simple attributes (or simple components of composite attributes) of W as attributes. In addition, include as foreign key attributes of R the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s).

DEPENDENT

EMPL-SSN	NAME	Relationship

Step 3: For each **binary 1:1 relationship type** R in the ER schema, identify the relations S and T that correspond to the entity types participating in R. Choose one of the relations, say S, and include the primary key of T as a foreign key in S. Include all the simple attributes of R as attributes of S.

DEPARTMENT

MANAGER-SSN	StartDate

Step 4: For each regular **binary 1:N relationship type** R identify the relation (N) relation S. Include the primary key of T as a foreign key of S. Simple attributes of R map to attributes of S.

EMPLOYEE

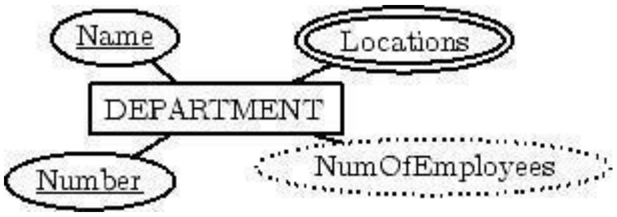
SupervisorSSN

Step 5: For each **binary M:N relationship type** R, create a relation S. Include the primary keys of participant relations as foreign keys in S. Their combination will be the primary key for S. Simple attributes of R become attributes of S.

WORKS-FOR

EmployeeSSN	DeptNumber

Step 6: For each **multi-valued attribute A**, create a new relation R. This relation will include an attribute corresponding to A, plus the primary key K of the parent relation (entity type or relationship type) as a foreign key in R. The primary key of R is the combination of A and K.



DEP-LOCATION

Location	DEP-NUMBER

Step 7: For each **n-ary relationship type R**, where n>2, create a new relation S to represent R. Include the primary keys of the relations participating in R as foreign keys in S. Simple attributes of R map to attributes of S. The primary key of S is a combination of all the foreign keys that reference the participants that have cardinality constraint > 1.

For a recursive relationship, we will need a new relation.