Unit 2: The Extended Entity Relationship Model and Object Model

The ER model

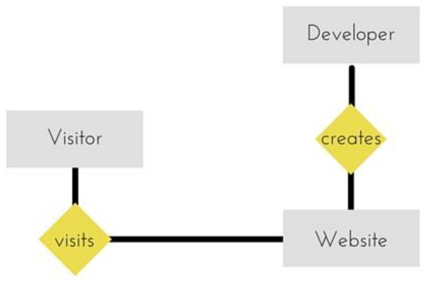
The entity-relationship model (or ER model) is a way of graphically representing the logical relationships of entities (or objects) in order to create a database**.**

In ER modeling, the structure for a database is described as a diagram, called an entity-relationship diagram (or ER diagram), that resembles the graphical breakdown of a sentence into its grammatical parts. Entities are rendered as points, polygons, circles, or ovals. Relationships are portrayed as lines connecting the points, polygons, circles, or ovals. Any ER diagram has an equivalent relational table, and any relational table has an equivalent ER diagram. ER diagramming is an invaluable aid to engineers in the design, optimization, and debugging of database programs.

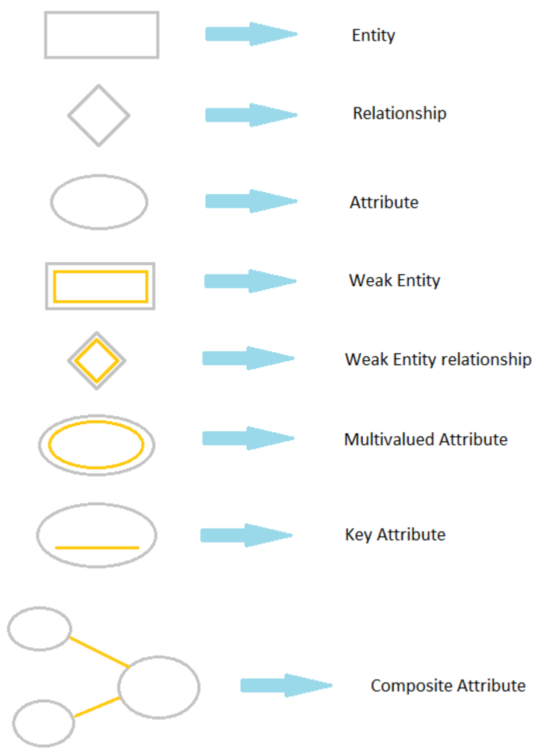
An entity is a real-world item or concept that exists on its own. For example employees, departments, products, or networks. An entity can be defined by means of its properties, called attributes. Relationships are the equivalent of verbs or associations, such as the act of purchasing, the act of repairing, being a member of a group, or being a supervisor of a department. A relationship can be defined according to the number of entities associated with it, known as the degree.

Creation of an ER diagram, which is one of the first steps in designing a database, helps the designer(s) to understand and to specify the desired components of the database and the relationships among those components. An ER model is a diagram containing entities or "items", relationships among them, and attributes of the entities and the relationships.

ER-Diagram is a visual representation of data that describes how data is related to each other.



Symbols and notations used in ER Diagram

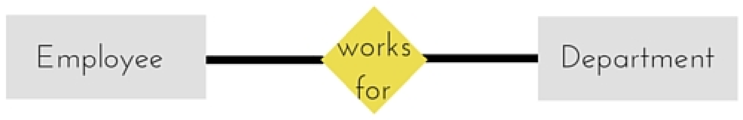


<u>Components of E-R Diagram</u>

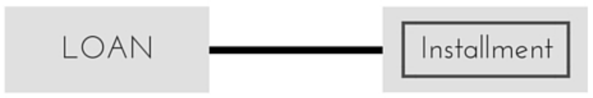
The E-R diagram has three main components.

1) Entity

An Entity can be any object, place, person or class. In E-R Diagram, an entity is represented using rectangles. Consider an example of an Organization. Employee, Manager, Department, Product and many more can be taken as entities from an Organization. The basic object that the ER model represents is an entity, which is a "thing" in the real world with an independent existence.



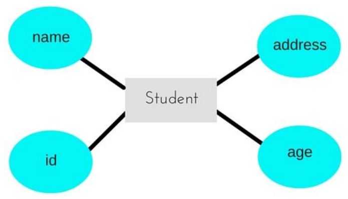
<u>Weak Entity</u>
Weak entity is an entity that depends on another entity. Weak entity doen't have key attribute of their own. Double rectangle represents weak entity.

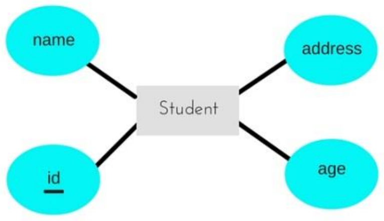


2) Attribute
An Attribute describes a property or characteristic of an entity. For example, Name, Age, Address etc. can be attributes of a Student. An attribute is represented using eclipse.

<u>Key Attribute</u>

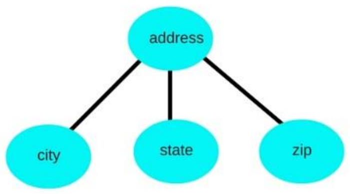
Key attribute represents the main characteristic of an Entity. It is used to represent Primary key. Ellipse with underlying lines represent Key Attribute.





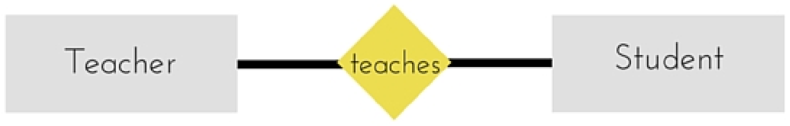
Composite Attribute

An attribute can also have their own attributes. These attributes are known as Composite attribute.



3) Relationship

A Relationship describes relations between entities. Relationship is represented using diamonds.

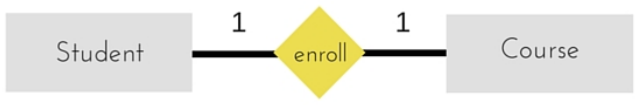


There are three types of relationship that exist between Entities.

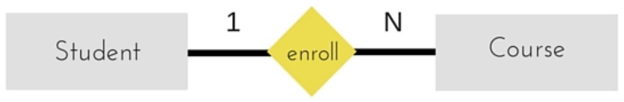
- Binary Relationship
- Recursive Relationship
- Ternary Relationship

Binary Relationship
Binary Relationship means relation between two Entities. This is further divided into three types.

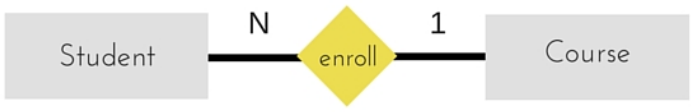
1. **One to One**: This type of relationship is rarely seen in real world.



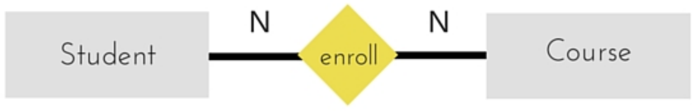
2. **One to Many**: It reflects business rule that one entity is associated with many number of same entity. The example for this relation might sound a little weird, but this means that one student can enroll too many courses, but one course will have one Student.



3. **Many to One:** It reflects business rule that many entities can be associated with just one entity. For example, Student enrolls for only one Course but a Course can have many Students.

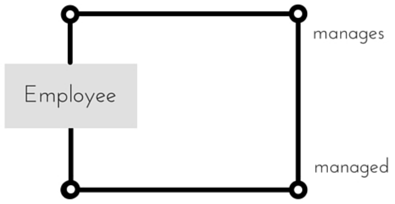


4. **Many to Many :**



Recursive Relationship

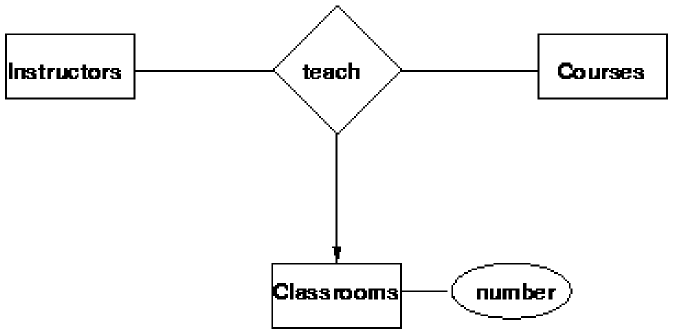
When an Entity is related with itself it is known as Recursive Relationship.



Employee may manage other employee. Each employee is managed by at most one employee.

Ternary Relationship/ Multi way relationship

Relationship of degree three is called Ternary relationship.



For 1 instructor and 1 course, there can be *at most* 1 classroom

Motivation for complex data types

Most relational DBMSs support only a few data types. Many business applications require large amounts of complex data such as images, audio, and video. Integration of complex data with simple data drives the demand for object database technology.

Database management systems traditionally dealt with simple tabular data. In recent years, object-relational database systems (ORDBMSs) were designed to support complex data types. Images, videos, and textual objects have been explicitly mentioned as examples of the data types ORDBMSs are intended to support. Nonetheless, current database systems have a long way to go before they can support such complex data types satisfactorily. In the context of text and XML data, challenges include efficient support for searches over textual content and support for searches that exploit the loose structure of XML data.

Traditional database applications have conceptually simple data types. The basic data items are records that are fairly small and whose fields are atomic—that is, they are not further structured, and first normal form holds (see Chapter 8). Further, there are only a few record types.

In recent years, demand has grown for ways to deal with more complex data types. Consider, for example, addresses. While an entire address could be viewed as an atomic data item of type string, this view would hide details such as the street address, city, state, and postal code, which could be of interest to queries. On the other hand, if an address were represented by breaking it into the components (street address, city, state, and postal code), writing queries would be more complicated since they would have to mention each field. A better alternative is to allow structured data types that allow a type *address* with subparts *street_address*, *city*, *state*, and *postal_code*.

As another example, consider multivalued attributes from the E-R model. Such attributes are natural, for example, for representing phone numbers, since people

title	*author_array*	*publisher* (*name*, *branch*)	*keyword_set*
Compilers	[Smith, Jones]	(McGraw-Hill, NewYork)	{parsing, analysis}
Networks	[Jones, Frick]	(Oxford, London)	{Internet, Web}

Figure 22.1 Non-1NF books relation, *books*.

✓

may have more than one phone. The alternative of normalization by creating a new relation is expensive and artificial for this example.

With complex type systems we can represent E-R model concepts, such as composite attributes, multivalued attributes, generalization, and specialization directly, without a complex translation to the relational model.

In Chapter 8, we defined *first normal form* (1NF), which requires that all attributes have *atomic domains*. Recall that a domain is *atomic* if elements of the domain are considered to be indivisible units.

The assumption of 1NF is a natural one in the database application examples we have considered. However, not all applications are best modeled by 1NF relations. For example, rather than view a database as a set of records, users of certain applications view it as a set of objects (or entities). These objects may require several records for their representation. A simple, easy-to-use interface requires a one-to-one correspondence between the user's intuitive notion of an object and the database system's notion of a data item.

Consider, for example, a library application, and suppose we wish to store the following information for each book:

- Book title.
- List of authors.
- Publisher.
- Set of keywords.

We can see that, if we define a relation for the preceding information, several domains will be nonatomic.

- **Authors**. A book may have a list of authors, which we can represent as an array. Nevertheless, we may want to find all books of which Jones was one of the authors. Thus, we are interested in a subpart of the domain element "authors."
- **Keywords**. If we store a set of keywords for a book, we expect to be able to retrieve all books whose keywords include one or more specified keywords. Thus, we view the domain of the set of keywords as nonatomic.
- **Publisher**. Unlike *keywords* and *authors*, *publisher* does not have a set-valued domain. However, we may view *publisher* as consisting of the subfields *name* and *branch*. This view makes the domain of *publisher* nonatomic.

Figure 22.1 shows an example relation, *books*.

title	*author*	*position*
Compilers	Smith	1
Compilers	Jones	2
Networks	Jones	1
Networks	Frick	2

authors

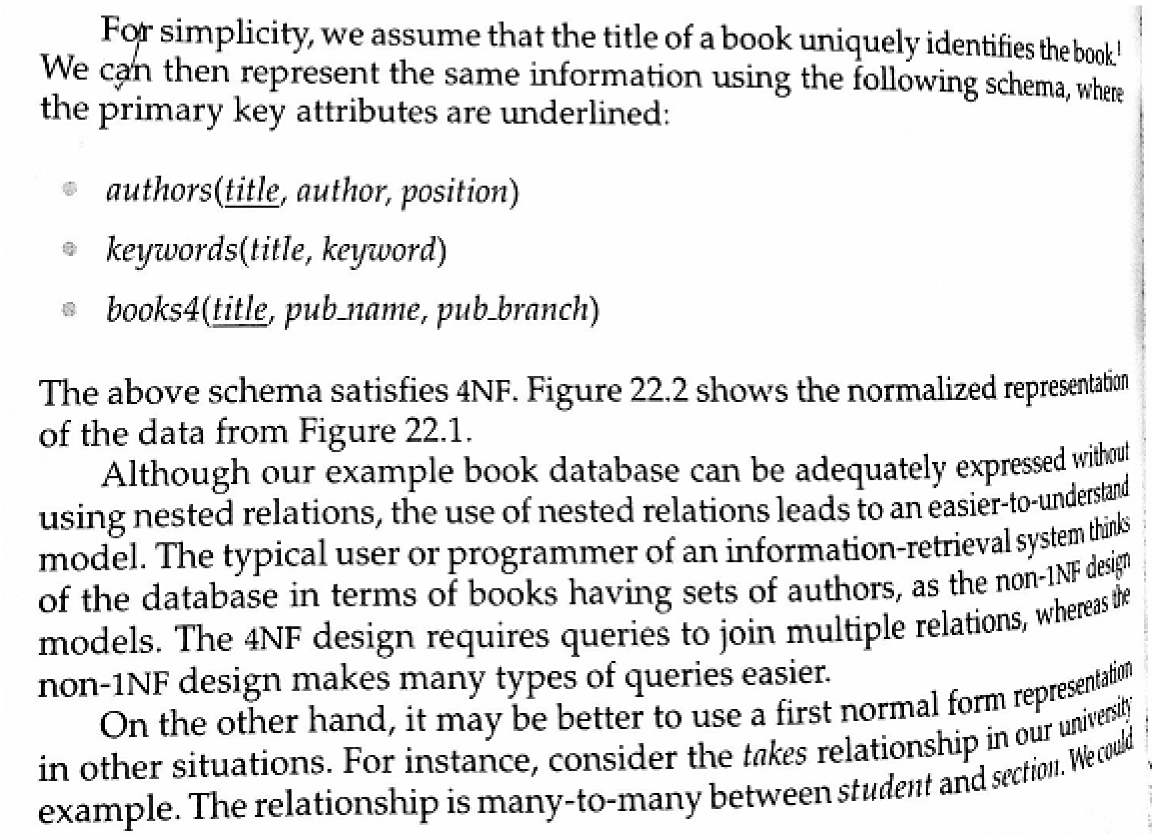
title	*keyword*
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

title	*pub_name*	*pub_branch*
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4

Figure 22.2 4NF version of the relation *books*.



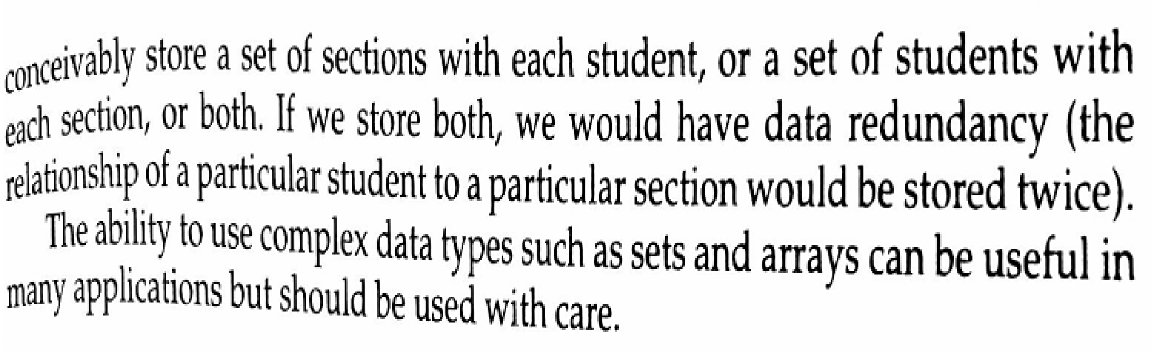
For simplicity, we assume that the title of a book uniquely identifies the book! We can then represent the same information using the following schema, where the primary key attributes are underlined:

- *authors*(*title*, *author*, *position*)
- *keywords*(*title*, *keyword*)
- *books4*(*title*, *pub_name*, *pub_branch*)

The above schema satisfies 4NF. Figure 22.2 shows the normalized representation of the data from Figure 22.1.

Although our example book database can be adequately expressed without using nested relations, the use of nested relations leads to an easier-to-understand model. The typical user or programmer of an information-retrieval system thinks of the database in terms of books having sets of authors, as the non-1NF design models. The 4NF design requires queries to join multiple relations, whereas the non-1NF design makes many types of queries easier.

On the other hand, it may be better to use a first normal form representation in other situations. For instance, consider the *takes* relationship in our university example. The relationship is many-to-many between *student* and *section*. We could



conceivably store a set of sections with each student, or a set of students with each section, or both. If we store both, we would have data redundancy (the relationship of a particular student to a particular section would be stored twice).

The ability to use complex data types such as sets and arrays can be useful in many applications but should be used with care.

User defined abstract data types and structured types

Abstract Data Types (ADTs) are user defined data types. Unlike traditional database management system which only provide for primitive data types such as INTEGER and CHARACTER, the object-oriented programming languages allow for the creation of abstract data types. The data types offered in commercial database systems, CHAR INTEGER NUMBER, VARCHAR, BIT, are sufficient for most relational database applications but developers are now beginning to realize that the ability to create user-defined data types can greatly simplify their database design.

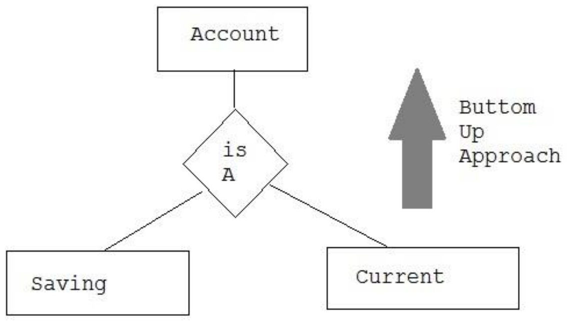
A data type can be considered abstract when it is defined in terms of operations on it, and its implementation is hidden (so that we can always replace one implementation with another for, e.g., efficiency reasons, and this will not interfere with anything in the program).

Specialization and Generalization

The ER Model has the power of expressing database entities in a conceptual hierarchical manner. As the hierarchy goes up, it generalizes the view of entities, and as we go deep in the hierarchy, it gives us the detail of every entity included.

Generalization:

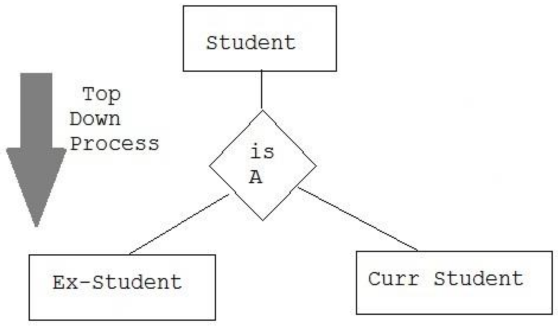
Generalization is a bottom-up approach in which two lower level entities combine to form a higher level entity.



Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass. Shared characteristics can be attributes, associations, or methods

Specialization:

Specialization is opposite to Generalization. It is a top-down approach in which one higher level entity can be broken down into two lower level entity. In specialization, some higher level entities may not have lower-level entity sets at all.



Specialization is the result of taking a subset of higher level entity set to form a lower entity set. It is the top down design process. Ie. We introduce subgrouping within an entity set that are different from other entity in the set. For instance, a subset of entities within an entity set may have attributes that are not shared by all the entities in the entity set.

A lower level entity set inherits all the attributes and relationship participation of higher level entity set to which it is linked. Following figure shows the concept of specialization and generalization.

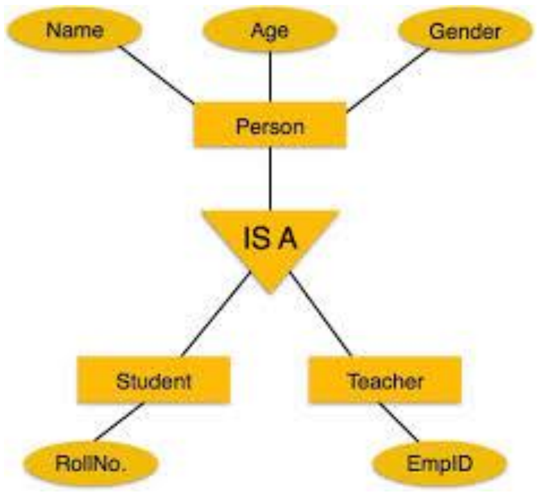
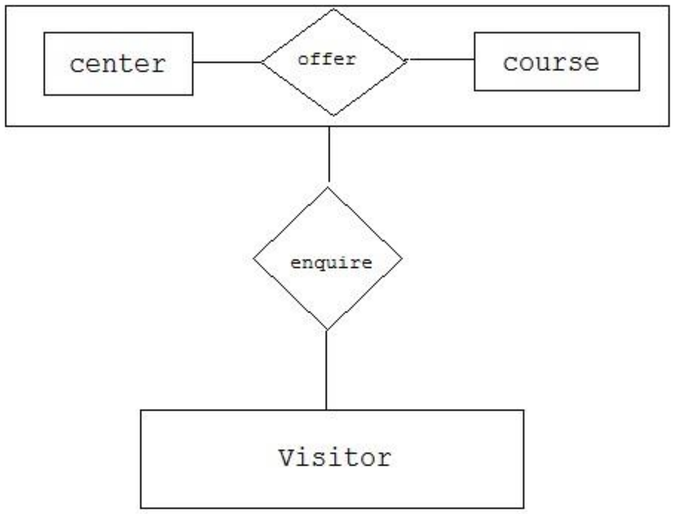


Fig : Generalization and specialization

The attributes of a Person class such as name, age, and gender can be inherited by lower-level entities such as Student or Teacher.

Aggregation:

Aggregation is a process when relation between two entities is treated as a single entity. Here the relation between Center and Course, is acting as an Entity in relation with Visitor.



Relationship types of degree higher than two

Relationship type of degree two- binary relationship and a relationship type of degree three- ternary relationship.

3.9 Relationship Types of Degree Higher Than Two

In Section 3.4.2 we defined the **degree** of a relationship type as the number of participating entity types and called a relationship type of degree two *binary* and a relationship type of degree three *ternary*. In this section, we elaborate on the differences between binary and higher-degree relationships, when to choose higher-degree or binary relationships, and constraints on higher-degree relationships.

3.9.1 Choosing between Binary and Ternary (or Higher-Degree) Relationships

The ER diagram notation for a ternary relationship type is shown in Figure 3.17(a), which displays the schema for the SUPPLY relationship type that was displayed at the instance level in Figure 3.10. Recall that the relationship set of SUPPLY is a set of relationship instances (s, j, p), where s is a SUPPLIER who is currently supplying a PART p to a PROJECT j. In general, a relationship type R of degree n will have n edges in an ER diagram, one connecting R to each participating entity type.

Figure 3.17(b) shows an ER diagram for the three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. In general, a ternary relationship type represents different information than do three binary relationship types. Consider the three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. Suppose that CAN_SUPPLY, between SUPPLIER and PART, includes an instance (s, p) whenever supplier s *can supply* part p (to any project); USES, between PROJECT and PART, includes an instance (j, p) whenever project j uses part p; and SUPPLIES, between SUPPLIER and PROJECT, includes an instance (s, j) whenever supplier s supplies some part to project j. The existence of three relationship instances (s, p), (j, p), and (s, j) in CAN_SUPPLY, USES, and SUPPLIES, respectively, does not necessarily imply

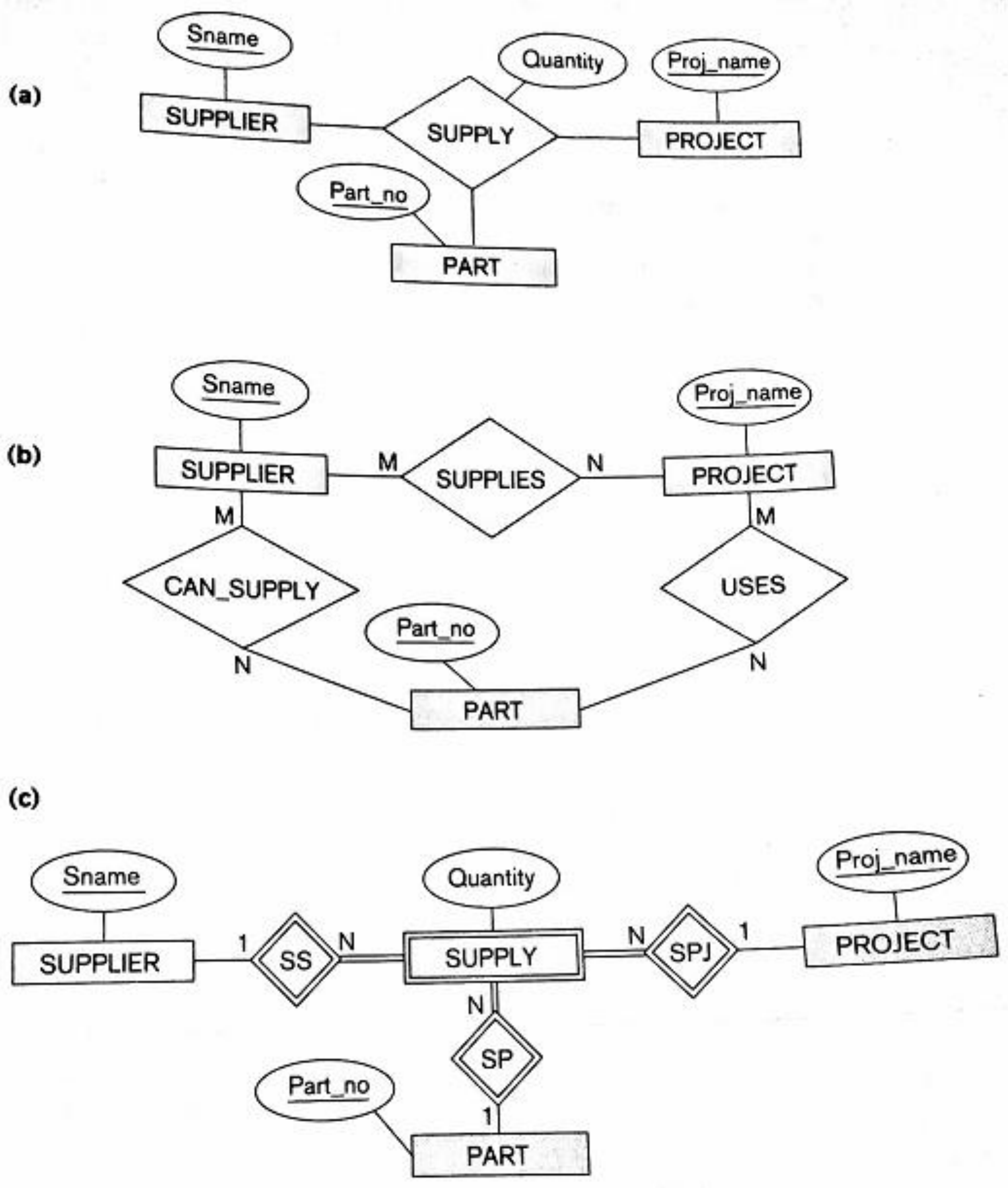


Figure 3.17
Ternary relationship types. (a) The SUPPLY relationship. (b) Three binary relationships not equivalent to SUPPLY. (c) SUPPLY represented as a weak entity type.

that an instance (s, j, p) exists in the ternary relationship SUPPLY, because the *meaning is different*. It is often tricky to decide whether a particular relationship should be represented as a relationship type of degree n or should be broken down into several relationship types of smaller degrees. The designer must base this decision on the semantics or meaning of the particular situation being represented. The typical solution is to include the ternary relationship *plus* one or more of the binary relationships, if they represent different meanings and if all are needed by the application.

Some database design tools are based on variations of the ER model that permit only binary relationships. In this case, a ternary relationship such as SUPPLY must be represented as a weak entity type, with no partial key and with three identifying relationships. The three participating entity types SUPPLIER, PART, and PROJECT are together the owner entity types (see Figure 3.17(c)). Hence, an entity in the weak entity type SUPPLY of Figure 3.17(c) is identified by the combination of its three owner entities from SUPPLIER, PART, and PROJECT.

It is also possible to represent the ternary relationship as a regular entity type by introducing an artificial or surrogate key. In this example, a key attribute Supply_id could be used for the supply entity type, converting it into a regular entity type. Three binary 1:N relationships relate SUPPLY to the three participating entity types.

Another example is shown in Figure 3.18. The ternary relationship type OFFERS represents information on instructors offering courses during particular semesters; hence it includes a relationship instance (*i*, *s*, *c*) whenever INSTRUCTOR *i* offers COURSE *c* during SEMESTER *s*. The three binary relationship types shown in Figure 4.12 have the following meanings: CAN_TEACH relates a course to the instructors who *can teach* that course, TAUGHT_DURING relates a semester to the instructors who *taught some course* during that semester, and OFFERED_DURING relates a semester to the courses offered during that semester *by any instructor*. These ternary and binary relationships represent different information, but certain constraints should hold among the relationships. For example, a relationship instance (*i*, *s*, *c*) should not exist in OFFERS *unless* an instance (*i*, *s*) exists in TAUGHT_DURING, an instance (*s*, *c*) exists in OFFERED_DURING, and an instance (*i*, *c*) exists in CAN_TEACH. However, the reverse is not always true; we may have instances (*i*, *s*), (*s*, *c*), and (*i*, *c*) in the three binary relationship types with no corre-

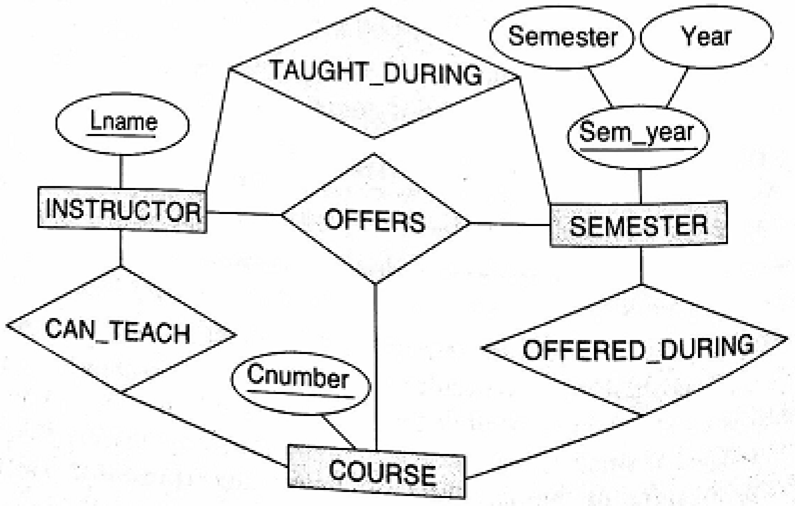


Figure 3.18
Another example of ternary versus binary relationship types.

sponding instance (*i*, *s*, *c*) in OFFERS. Note that in this example, based on the meanings of the relationships, we can infer the instances of TAUGHT_DURING and OFFERED_DURING from the instances in OFFERS, but we cannot infer the instances of CAN_TEACH; therefore, TAUGHT_DURING and OFFERED_DURING are redundant and can be left out.

Although in general three binary relationships *cannot* replace a ternary relationship, they may do so under certain *additional constraints*. In our example, if the CAN_TEACH relationship is 1:1 (an instructor can teach one course, and a course can be taught by only one instructor), then the ternary relationship OFFERS can be left out because it can be inferred from the three binary relationships CAN_TEACH, TAUGHT_DURING, and OFFERED_DURING. The schema designer must analyze the meaning of each specific situation to decide which of the binary and ternary relationship types are needed.

Notice that it is possible to have a weak entity type with a ternary (or *n*-ary) identifying relationship type. In this case, the weak entity type can have *several* owner entity types. An example is shown in Figure 3.19.

3.9.2 Constraints on Ternary (or Higher-Degree) Relationships

There are two notations for specifying structural constraints on *n*-ary relationships, and they specify different constraints. They should thus *both be used* if it is important to fully specify the structural constraints on a ternary or higher-degree relationship. The first notation is based on the cardinality ratio notation of binary relationships displayed in Figure 3.2. Here, a 1, M, or N is specified on each participation arc (both M and N symbols stand for *many* or *any number*).[15] Let us illustrate this constraint using the SUPPLY relationship in Figure 3.17.

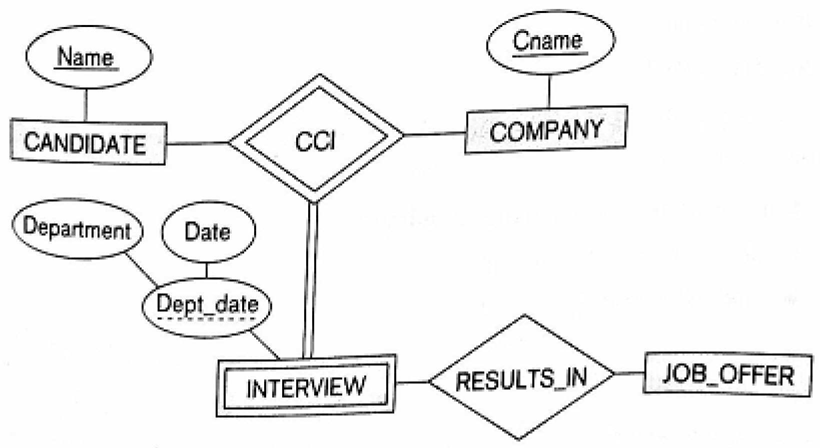


Figure 3.19
A weak entity type INTERVIEW with a ternary identifying relationship type.

Relational database design by EER-to-relational mapping

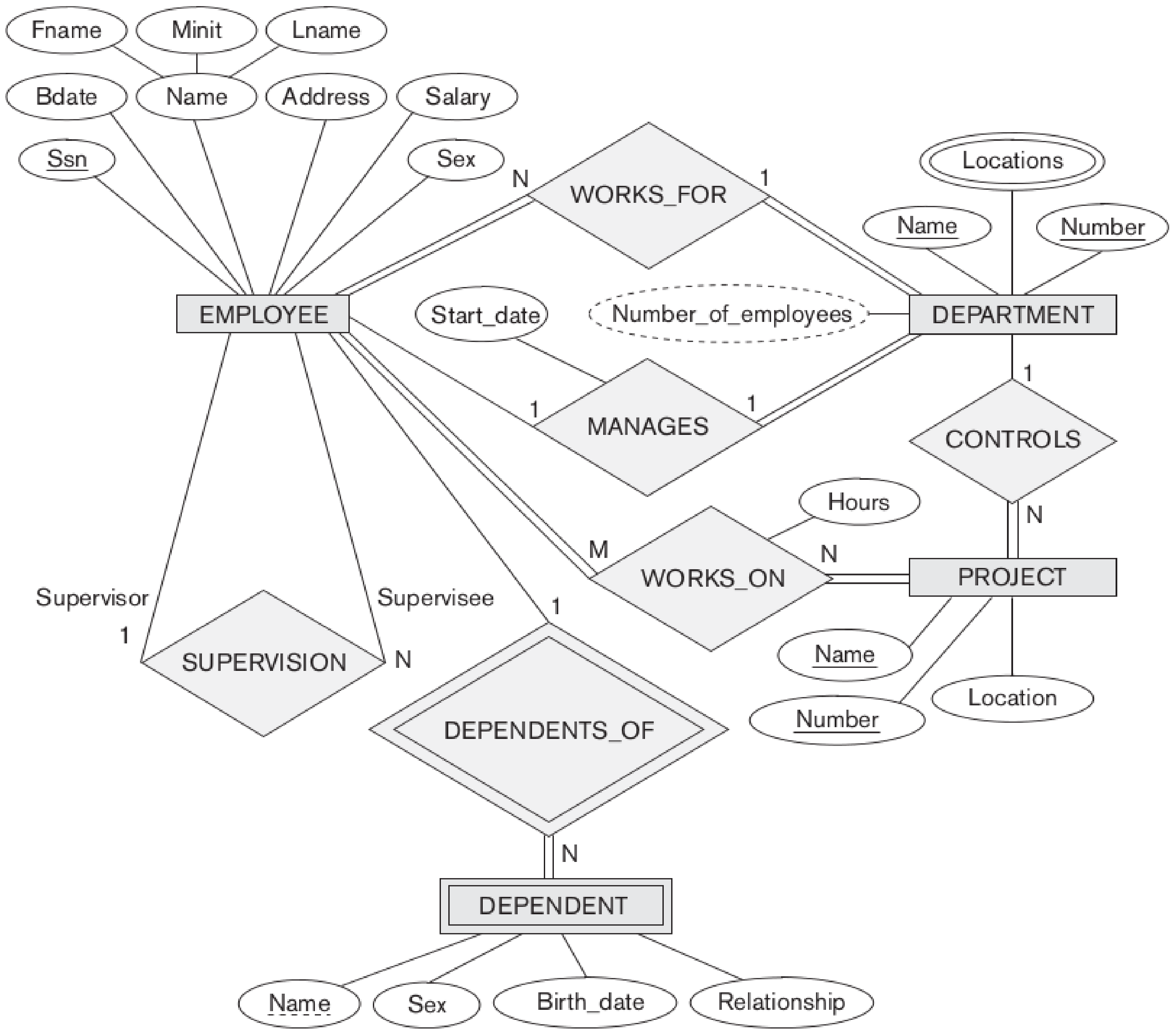
This focuses on designing a relational database schema based on a conceptual schema design.

Logical database design relates the concepts of the Entity-Relationship (ER) and Enhanced-ER (EER) models

ER-to-Relational Mapping Algorithm

We use the COMPANY database example to illustrate the mapping procedure. The COMPANY ER schema is shown below and the corresponding COMPANY relational database schema is shown after the COMPANY ER diagram to illustrate the mapping steps.

Figure 9.1
The ER conceptual schema diagram for the COMPANY database.



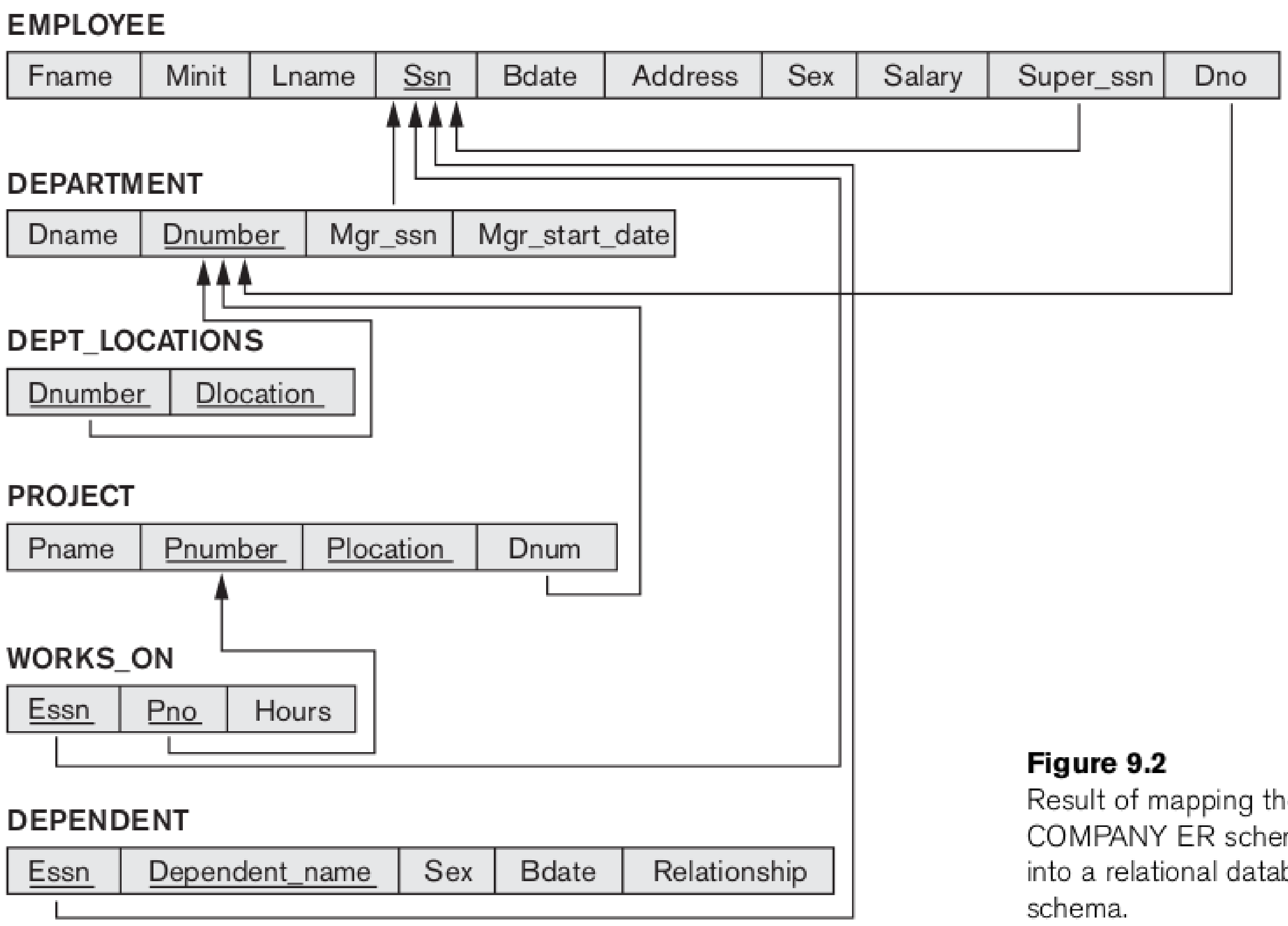


Figure 9.2
Result of mapping the COMPANY ER schema into a relational database schema.

In Detail:

Step 1: Regular Entity Types

Create an entity relation for each strong entity type. Include all single-valued attributes. Flatten composite attributes. Keys become secondary keys, except for the one chosen to be the primary key.

Step 2: Weak Entity Types

Also create an entity relation for each weak entity type, similarly including its (flattened) single-valued attributes. In addition, add the primary key of each owner entity type as a foreign key attribute here. Possibly make this foreign key CASCADE.

Step 3: Binary 1:1 Relationship Types

Let the relationship be of the form [S]——<R>——[T].

1. Foreign key approach: The primary key of T is added as a foreign key in S. Attributes of R are moved to S (possibly renaming them for clarity). If only one of the entities has total participation it's better to call it S, to avoid null attributes. If neither entity has total participation nulls may be unavoidable. This is the preferred approach in typical cases.
2. Merged relation approach: Both entity types are stored in the same relational table, “pre-joined”. If the relationship is not total both ways, there will be null padding on tuples that represent just one entity type. Any attributes of R are also moved to this table.

3. Cross-reference approach: A separate relation represents R; each tuple is a foreign key from S and a foreign key from T. Any attributes of R are also added to this relation. Here foreign keys should CASCADE.

Step 4: Binary 1:N Relationship Types

Let the relationship be of the form [S]——^{N}<R>^{1}——[T]. The primary key of T is added as a foreign key in S. Attributes of R are moved to S. This is the foreign key approach. The merged relation approach is not possible for 1:N relationships. (Why?) The cross-reference approach might be used if the join cost is worth avoid null storage.

Step 5: Binary M:N Relationship Types

Here the cross-reference approach (also called a relationship relation) is the only possible way.

Step 6: Multivalued Attributes

Let an entity S have multivalued attribute A. Create a new relation R representing the attribute, with a foreign key into S added. The primary key of R is the combination of the foreign key and A. Once again this relation is dependent on an “owner relation” so its foreign key should CASCADE.

Step 7: Higher-Arity Relationship Types

Here again, use the cross-reference approach. For each n-ary relationship create a relation to represent it. Add a foreign key into each participating entity type. Also add any attributes of the relationship. The primary key of this relation is the combination of all foreign keys into participating entity types that do not have a max cardinality of 1.

Summary:

ER Model	**Relational Model**
Entity type	Entity relation
1:1 and 1:N relationship type	Foreign key or relationship relation
M:N relationship type	Relationship relation and two foreign keys
n-ary relationship type	Relationship relation and n foreign keys
Simple attribute	Attribute
Composite attribute	Set of component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary key or secondary key

Basic concepts on UML

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

- UML stands for Unified Modeling Language.
- UML is different from the other common programming languages like C++, Java, COBOL etc.
- UML is a pictorial language used to make software blue prints.

So UML can be described as a general purpose visual modeling language to visualize, specify, construct and document software system. Although UML is generally used to model software systems but it is not limited within this boundary. It is also used to model non software systems as well like process flow in a manufacturing unit etc.

UML is not a programming language but tools can be used to generate code in various languages using UML diagrams. UML has a direct relation with object oriented analysis and design. After some standardization UML is become an OMG (Object Management Group) standard.

Goals of UML:

A picture is worth a thousand words, this absolutely fits while discussing about UML. Object oriented concepts were introduced much earlier than UML. So at that time there were no standard methodologies to organize and consolidate the object oriented development. At that point of time UML came into picture.

There are a number of goals for developing UML but the most important is to define some general purpose modeling language which all modelers can use and also it needs to be made simple to understand and use.

UML diagrams are not only made for developers but also for business users, common people and anybody interested to understand the system. The system can be a software or non-software. So it must be clear that UML is not a development method rather it accompanies with processes to make a successful system.

At the conclusion the goal of UML can be defined as a simple modeling mechanism to model all possible practical systems in today's complex environment.

There are two categories of diagrams in UML

1. Structural Diagrams

The structural diagrams represent the static aspect of the system. These static aspects represent those parts of a diagram which forms the main structure and therefore stable.

These static parts are represents by classes, interfaces, objects, components and nodes. The four structural diagrams are:

~ Class diagram
~ Object diagram
~ Component diagram
~ Deployment diagram

2. Behavioral Diagrams

Behavioral diagrams basically capture the dynamic aspect of a system. Dynamic aspect can be further described as the changing/moving parts of a system.

UML has the following five types of behavioral diagrams:

~ Use case diagram
~ Sequence diagram
~ Collaboration diagram
~ State chart diagram
~ Activity diagram