Chapter 4
Conceptual Modeling

We got a “clear” picture of the structure of an RDB from last chapter. The question is how could we get there from a project description?

For example, assume that we have understood what should be done with the Registration database, how should we design the database schema, i.e., tables, indices, domains, assertions (ICs), triggers, etc., in terms of a bunch of Create statements?

We will show, in this chapter, how to visually come up with a preliminary design, in terms of a semantic model, by following the Entity-relationship (ER) approach. (Have you ever played with MS Visio?)

We will then continue with an incremental improvement, guided by the normalization theory, as discussed in Unit 7 to finish the database design.
The E/R approach

The entity-relationship approach has nothing to do with the relational data modeling or any data modeling. It is just a methodology, which can help us to come up with a (relational) data model.

This approach is involved with two components, *entity* and *relationship*. Entity represents the objects we will deal with, such as students, professors, courses, etc.; and relationship represents the connection among those entities, such as a student has to take courses, which leads to an relation between those two objects.

There could also be some constraints on such relationship, such as a student cannot take two courses scheduled at the same time.

An *E/R diagram* is a graphical representation of all the objects and their constrained relationships.
Then what?

Once the design part is over with an E/R diagram as the end product, we can apply some standard procedure to convert it into a set of **Create table** statements that lead to a database, *although some aspects of the diagram can’t be mechanically converted over.*

The creative aspect of the E/R approach lies in the process of deciding which entities, relationship, and constraints to use when modeling an enterprise.

A designer must have a thorough understanding of the enterprise to come up with a good design. (Project Phase II)

Following the E/R approach, a designer only cares about the *what* part, but not *how* this design will work out, particularly the efficiency part.
Entities

What we do first is to use entities to represent the objects in the enterprise that we are trying to model.

An entity can be defined as a thing, which can be distinctly identified. It can be as concrete as someone’s name, e.g., John Doe, or a bank account, e.g., 1234567. Those concrete values can be aggregated to entity type, just as objects can be aggregated into class in Java.

Thus, we can have such entities as Professors, Addresses, Courses, etc..

An entity typically has attributes, each of them specifies a certain property of that entity. For example, a professor has her name, maybe an ID, an office, and a bunch of courses she teaches for the semester, etc.. Such properties are represented by associating a value for an attribute, taken from a certain domain.
A couple of points

1. Although two entities with the same type could have identical values, there is no point of doing that. (Set)

2. We could group attributes of different stuff such as car and people into one entity type, there is again no point of doing that. In practice, you only group similar stuff, or semantically related, together. (Minimality)

3. An attribute value can be a set. For example, childrenNames can be an attribute. This does cause certain issue, and leads to some extra effort in its implementation.

4. Any entity type has a key as specified by a key constraint (uniqueness and minimality).
5. We use the *schema of an entity type* to refer to the name of the type, the collection of its attributes, their respective domains, and value indicators (where it is a set or not), together with all its constraints. (Sounds like a table structure?)

6. An *E/R digram* is the end result of following the E/R modeling approach. Below shows an example.

In the above, **SSN** indicates that it is the key, and the double shelled egg indicated **Hobby** could have a set as its value.
Relationships

Relationship is the other component of the E/R modeling process, a much more complicated one.

A relationship is defined as an association among entities. For example, all the Student entities are related to the Program entities, via a relationship type of MajorsIn. More specifically, under this relationship, Mary might be related to CS major, while John might be to IT.

The entities involved in a relationship are called participants of that relationship.

While an entity is implemented as a table, a relationship can be implemented as (an) attribute(s) or even constraints, instead of a relation (table).
A couple of points

1. A relationship can also have attributes. For example, since about half of the students change their majors at least once, it makes sense for MajorsIn to have an attribute of Since.

2. A relationship needs other stuff, besides attributes. For example, given an entity type Employee, a relationship ReportsTo, and a pair (Tom, Jane), we still don't know who reports to whom.

It is actually not enough to split the Employee type to Subordinate and Supervisor, since someone can be both a subordinate and a supervisor.

A solution is to specify a different role to the participants of the ReportsTo type. Thus, in a particular relationship instance, the role of John is supervisor, while in another one, he might be a subordinator.
3. When all the entities involved belong to different entity types, we don’t need to explicitly label the roles. Otherwise, we have to. Below show some examples.

**Question:** Why is this triplet used as the key?
A bit more formally,...

A schema of a relationship type includes the following:

1. A list of attributes with the respective domains. An attribute again can have a single value or a set as its value.

2. A list of roles with their respective entity types. A role always has an entity type as its value.

The number of roles as contained in a relationship type is called its degree.

3. A set of constraints, which we will discuss more fully later.
4. A relationship type $R$ of degree $n$ is defined by its attributes $A_1, A_2, \ldots, A_k$ and roles $R_1, R_2, \ldots, R_n$. Any of its instance is defined as $(e_1, \ldots, e_n; a_1, \ldots, a_k)$, where $e_i$ is an entity with role $R_i$, and $a_j$ is a value of attribute $A_j$.

For example, the relationship MajorsIn can have the schema (Student, Program; Since), and one of its instance might be ('Homer Simpson', EE; 1994).

The above states that Homer Simpson became an EE major since 1994. In particular, “Homer Simpson” is an entity with Student being its role.
More about keys

The key of a relationship allows the designer to express many constraints she wants to put on the design.

A key of a relationship $R$ has to be defined as a minimal set of roles and attributes of $R$ whose instance uniquely identifies the relationship instances of $R$.

Let $R_1, \ldots, R_k$ be a subset of all the roles of $R$, and let $A_1, \ldots, A_s$ be a subset of all the attributes of $R$, then $(R_1, \ldots, R_k; A_1, \ldots, A_s)$ is a key of $R$ if both of the following two conditions are met: 1. Uniqueness: $R$ does not have a pair of distinct rows sharing the same value for $(R_1, \ldots, R_k; A_1, \ldots, A_s)$.

2. Minimality: No proper subset of $(R_1, \ldots, R_k; A_1, \ldots, A_s)$ is a key of $R$. 
Graphically speaking,...

For the WorksIn relation, it is reasonable to assume that there are multiple professors in each department. Assume each professor works for at most one department, we can use Professor alone as the key of this type, as indicated by an arrow.

A key consisting of several roles can be so indicated with text, such as that for the Sold type.

This key implies that a customer can buy the same product only once on any given day. 😞
Cardinality constraints

Single-role key constraints, drawn as an arrow, can be generalized using the notion of cardinality constraints.

Let $C$ be an entity type, connected to a relationship $A$ via a role $R$.

A cardinality constraint on $R$ can be represented as a range $\min..\max$ attached to $R$, which places a restriction on the number of instances of $A$ in which a single $C$ instance can participate in role $R$. Below shows the same $C$ value can occur either once or twice in a tuple.
Why is it more general?

In the case of a key, the number has to be 0..1, meaning an entity instance participates in a relationship instance at most once.

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>John Doe</td>
<td>123 Main St.</td>
<td>Freshman</td>
</tr>
<tr>
<td>2222</td>
<td>Mary Smith</td>
<td>1 Lake St.</td>
<td>Freshman</td>
</tr>
<tr>
<td>1234</td>
<td>Joe Blow</td>
<td>6 Yard Ct.</td>
<td>Junior</td>
</tr>
</tbody>
</table>

Those three show up once, and all the other ids do not show up.
Even more general...

Below shows some more general stuff, where ‘*’ means 0..*. At most one C and D instance can participate in an instance of the relationship B; while multiple E and F instances can participate. In other words, there is a one-to-many correspondence from C to E and a many-to-many correspondence between E and F.

A professor (C) teaches multiple students (E), a lab assistant (D) can help in multiple courses (F); while one student can take multiple courses, and multiple students can take a course.
Entity type hierarchies

E/R diagram can be complicated and interesting since some of the relationship type can derived from other relationship types. For example, since every student is a person, Student is a derived type of Person, a base type. A derived type inherits all the attributes of the base type.

Therefore, every Person attribute is automatically a Student type. Particularly, the key for Person is also a key for Student.

Remember in Java, we have class childClass extends class parentClass, namely, childClass is derived from the parentClass. Thus, everything, fields and methods, of parentClass is inherited by the childClass.

Questions: is childClass a subclass of parentClass? Is its fields a sub set of that of parentClass?
The subtyping stuff sets up a relationship between the subtype in the role of Sub and the supertype in the role of Super. Such a relationship is often called the IsA relationship, and is used to constitute a classification hierarchy in the conceptual model.

Graphically speaking, this gives a way to draw the diagram in a more concise and precise manner.

Such a hierarchy naturally introduces many of the inheritance properties that we saw in the object oriented programming, as we mentioned about in terms of Java.
An example

Below shows an IsA hierarchy for some of the entities we have been discussing. Notice that freshman, sophomore, etc. are all instances of students.
Other constraints

The union of all of the freshman, sophomore, junior, and senior instances might equal that of the student instances. We thus might add a coverage constraint to the relevant IsA relationship.

On the other hand, they must be disjoint from each other, thus, we also add a disjoint constraint to this isA relationship.

The IsA relationship can also be used to address the issue of data partitioning. This might alleviate the data communication in a distributed environment for those data that are of local interest. If it is not needed elsewhere, why should we send it over?
Participation constraints

With the WorksIn relationship, we can only guarantee that one professor works in \textit{at most} one department, but we can’t rule out the possibility that a Professor instance is not participating in any WorksIn relationship instance. We can enforce such a participation that he works in \textit{at least} one department by placing a participation constraint.

Given an entity type $E$ and a relationship type $R$, a \textit{participation constraint} of $E$ in $R$ states that for $e \in E$, there is a $r \in R$, such that $e$ participates in $r$.

We thus can enforce, e.g., every student takes at least one course every semester.

We use a thick line to indicate such a constraint.
Graphically speaking,…

Below shows how to represent participating constraints in thick lines:

Notice a thick line with arrow represents the fact that a professor works in *exactly* (at least plus at most) one department.

While only the thick line without no arrow says that a student has to take at least one course, but she might take multiple.
Besides the IsA relationship, *part-of* is also useful, e.g., a wheel entity is part of a car entity.

In a *non-exclusive* relationship, the part can exist independently. For example, a tire can be part of a car, or ....

Similarly. a course might not be part of any major, e.g., CSDI 1400, or may be part of multiple majors, e.g., MA 2490/2500.
... and the other kind

In an *exclusive* relationship, the part cannot exist independently. For example, a program is part of a college. If the college closes, so do all the programs. When an employee is laid off, his dependents will all be taken off the system, as well.

Here, a *weak entity* is related to its *master entity*, through an *identifying relationship*.

The weak entities are represented using double boarded rectangles, as follows:
Notation summary

Below shows some of the notations we often use in drawing E/R diagrams.

Question: Now what?

Turn the graphics into tables....
From an E/R chart to RDB tables

We now have all the graphical stuff, the E/R diagrams, with which we want to derive a relational database, i.e., a bunch of integrated RDB tables.

It is straightforward to get the relations based on the entities. Each entity type is converted to a relation (table).

Assume that one instance is \((1111, \text{John Doe, 123 Main Street, \{Stamps, Coins\}}\), \((2222, \text{Mary Doe, 7 Lake Dr., \{Acting\}}\); then we can have three rows in the associated table as follows:

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>John Doe</td>
<td>123 Main St.</td>
<td>Stamps</td>
</tr>
<tr>
<td>1111</td>
<td>John Doe</td>
<td>123 Main St.</td>
<td>Coins</td>
</tr>
<tr>
<td>2222</td>
<td>Mary Doe</td>
<td>7 Lake Dr.</td>
<td>Acting</td>
</tr>
</tbody>
</table>

**Homework:** 4.2.
Candidate keys

When the entity type does not have set-valued attributes, each key of the entity type becomes a key of the corresponding relation schema. Otherwise, we have to throw in the attributes that have set values into the keys.

Thus, we can have the following statement for the above table.

Create table Person {
  SSN       Integer,
  Name      Char(20),
  Address   Char(50),
  Hobby     Char(10),
  Primary Key (SSN,Hobby))

Although it is doable, this shows that E/R does not lead to an ideal design of the database tables. We will talk about it further in the normalization chapter.
Relationships

The general procedure of converting a relationship type into relation schema(s) is as follows:

1. Identify the *attributes* of the relation for the relationship.

2. Identify the *candidate keys* of the relationship.

3. Identify the *foreign key* constraints.

Let's look into more details.
Step 1: Attributes

Given a relationship type $R$, the attributes of the corresponding RDB schema consist of the following:

a) the attributes of $R$; and

b) for each role in $R$, the primary key of the associated entity type, each of which also becomes a foreign key referencing the corresponding entity type.

Moreover, each and every of these primary keys must be declared not null for obvious reasons.
An issue

Attributes taken from the participants might have to be renamed to avoid ambiguity.

For example, in the marriage relationship, the primary key for both roles, Husband and Wife, could be (FirstName, FamilyName). Thus, when composing the relation type, there are two pairs of identical attributes. (FirstName, FirstName) and (FamilyName, FamilyName).

The schema might be the following:

Marriage{FirstName, FamilyName, FirstName, FamilyName, Date}

Does anybody understand it?
Make it more meaningful

For the first pair, they mean the first name of two different persons, thus have to be renamed to, e.g., (HusbandFirstName, WifeFirstName).

For the second pair, assuming both husband and wife share the same family name, it would simply be the family name of a married couple, thus we can simply delete the second occurrence.

The schema might be revised as the following:

```
Marriage{HusbandFirstName, WifeFirstName, 
    FamilyName, Date};
```

and the foreign keys

```
Marriage(HusbandFirstName) references Person(FirstName); 
Marriage(WifeFirstName) references Person(FirstName); 
Marriage(FamilyName) references Person(FamilyName);
```
Step 2: Candidate key

In most cases, the candidate keys of the relation schema can be derived using the candidate keys of participating entities.

For example, in the following diagram, the primary key of the worksIn relation will be \{Id\} of the Professor entity, assuming one professor can work in one department at a time.

If there exist set-value attributes for the relationship, we simply include them into a candidate key. Check out the example on page 27.
Step 3: Role and foreign key

In the E/R model, a role, when participating a relation, always refers to some entity that plays this role.

Thus, a role always translates to a foreign-key of the corresponding relationship, referring back to the entity.

A bit more formally, let \( R \) be a role in a relationship \( R \) that connects \( R \) to an entity type \( E \), and let \( rel(R) \) and \( rel(E) \) to denote the respective relation schemas derived from \( R \) and \( E \).

For each such role \( R \), its primary key, \( K \), is part of the primary key of \( E \), becomes a foreign key of \( rel(R) \) that references back to \( rel(E) \), provided that \( K \) is also a primary key of \( rel(E) \), as required by the FK constraint.
An example

Given the following diagram,

we can have the following specification:

Create table WorksIn (  
    Since Date, --attribute  
    ProfId Integer, --role  
    DeptId Char(4), --role  
    Primary key (ProfId), --primary key  
    Foreign key (ProfId) --role  
        references Professor (Id),  
    Foreign key (DeptId) --role  
        references Department))
Another example

Given the following diagram,

we can have the following specification:

Create table Sold (  
  Price Integer, --attribute  
  Date Date, --attribute  
  ProjId Integer, --role  
  SupplierId Integer, --role  
  PartNumber Integer, --role  
  Primary key (ProjId,PartNumber,Date),  
    --Date is a set based value  
  Foreign key (ProjId) references Project,  
    --ProjID is the primary key of Project  
  Foreign key (SupplierId) references Supplier (Id),  
    --SupplierId is not a primary key of Supplier  
    --Id is  
  Foreign key (PartNumber) references Part(Number))

**Question:** Why is SupplierId not part of the key?
Represent the IsA hierarchy

Since a key of any supertype is also a key of a subtype, in general, we can choose a candidate key of the supertype for all the subtypes related by the IsA hierarchy, by adding this key to each subtype, then convert the resulting entities into the relations.

For example, in the following hierarchy, we can choose SSN as the common key for all.
Represent the entities

We can then add this common key to all the related entities and come up with the following tables in the first round:

Person(SSN, Name, D.O.B)

Student(SSN, StartDate, GPA)

Freshman(SSN)

Sophomore(SSN, Major)

Junior(SSN, Major)

Senior(SSN, Advisor)

Employee(SSN, Department, Salary)

Secretary(SSN)

Technician(SSN, Specialization)
Parent and children

Moreover, we also need to add in the inclusion dependencies from subtypes to the supertype via a foreign key constraint so that every key instance for a subtype instance belongs to a supertype instance. This guarantees, e.g., every student is a person.

Thus, for the Student and the Employee types, a foreign key needs to be added as follows:

Foreign key (SSN) references Person

For the Freshman, etc.

Foreign key (SSN) references Student

For both the Technician and Secretary,

Foreign key (SSN) references Employee
Represent the Disjoint constraint

The IsA relationships, $C_1 \text{ IsA } C; \ldots, C_k \text{ IsA } C$, often satisfy the disjointness constraint, i.e., none of the respective instances share any values among them. In other words, every tuple in $C$ belongs to at most one category $C_i$, $i \in [1, k]$.

We can enforce this requirement by creating a summary relation for all the participating entities. The attributes of this summary relation is the union of the attributes of all the participating entities, plus one extra attribute indicating the original entity type of each tuple in the entity. This last entity can contain at most one value, enforcing the disjoint constraint.

We will also add in the common key as the primary key for all the tuples.

Finally, tuples that don’t have values for certain attributes will be assigned the null values.
An example tells all...

For the Freshman, Sophomore, Junior and Senior entities, connected to the Student entity with an IsA hierarchy satisfying a disjointness constraint, we can add a single relation schema, 
StudentSum(SSN, GPA, StartDate, Major, Advisor, Status),

with the value of Status being either Freshman, Sophomore, Junior, Or Senior.

This added Status guarantees, e.g., no student is both a freshman and a sophomore. (?)

As we mentioned that, for a Freshman type of row in the above relation, its Advisor attribute will be given a value of null.

We will also add two more relations EmployeeSum and PersonSum for the same reason.

Homework: 4.1, 4.3
Represent the Covering constraint

An IsA relationship, $C_1$ IsA $C$; ..., $C_k$ IsA $C$, also satisfies the covering constraint, i.e., the union of all the instances of $C_i$, $i \in [1, k]$, equals the collection of $C$. Thus, e.g., every student is either a freshman, a sophomore, a junior, or a senior.

To make sure this constraint is met, we can create one relation schema for each subtype of the IsA relationship, with the attributes for each such schema being the union of those in the subtype and those in the supertype, plus the common key.

We then remove the original super type $C$ since all its information now belong to the tables representing the subtypes.

For example, every person is kept in either the employee table or the student table. Thus, the table for Person can be deleted.
Another example is in order....

Assuming that the only people in our Person database are employees and students, then the covering constraint has to be satisfied for those three entity types.

Since both Student and Employee are subtypes and Person is the supertype, we can replace these three types with the following two tables:

\[ \text{EmplRel(SSN, Name, D.O.B., Department, Salary)} \]
\[ \text{StudRel(SSN, Name, D.O.B., GPA, StartDate)} \]

This guarantees that every person is either an employee or a student.

To prevent any inconsistency from happening because of redundancy, we remove the Person table.
A specific example

Consider the following instance of the Student entity:

<table>
<thead>
<tr>
<th>Id</th>
<th>SD</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>09/01/2013</td>
<td>3.2</td>
</tr>
<tr>
<td>2222</td>
<td>09/01/2014</td>
<td>3.8</td>
</tr>
</tbody>
</table>

an instance of the Freshman entity:

<table>
<thead>
<tr>
<th>Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>2222</td>
</tr>
</tbody>
</table>

and an instance of the Sophomore entity

<table>
<thead>
<tr>
<th>Id</th>
<th>Major</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>CS</td>
</tr>
</tbody>
</table>

**Question:** What could go wrong?

**Answer:** Disjointness and coverage.
To enforce the disjointness,...

namely, no student can be both a freshman and a sophomore, we come up with a StudSum entity, by merging the respective subtypes, Junior and Sophomore in this case, and the super type entities:

<table>
<thead>
<tr>
<th>Id</th>
<th>SD</th>
<th>GPA</th>
<th>Major</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>09/01/2013</td>
<td>3.2</td>
<td>CS</td>
<td>Sophomore</td>
</tr>
<tr>
<td>2222</td>
<td>09/01/2014</td>
<td>3.8</td>
<td>Null</td>
<td>Freshman</td>
</tr>
</tbody>
</table>

Since all the information as contained in both the Freshman and Sophomore entities have been included in this new StudSum entity, the original Freshman and Sophomore entities can be deleted.

Notice that this is a bottom-up process
To enforce the coverage, ... namely, every student has to be either a freshman or a sophomore. In place of the Student entity, we come up with two new entities: FreshRel

<table>
<thead>
<tr>
<th>Id</th>
<th>SD</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2222</td>
<td>09/01/2014</td>
<td>3.8</td>
</tr>
</tbody>
</table>

and SophoRel,

<table>
<thead>
<tr>
<th>Id</th>
<th>SD</th>
<th>GPA</th>
<th>Major</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>09/01/2013</td>
<td>3.2</td>
<td>CS</td>
</tr>
</tbody>
</table>

Since all the information as contained in the Student entity has been included in these two new entities, the original Student entity can be deleted.

Notice that this is a top-down process.
What have we got?

We will replace the original three entities with the following three: StudSum

<table>
<thead>
<tr>
<th>Id</th>
<th>SD</th>
<th>GPA</th>
<th>Major</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>09/01/2013</td>
<td>3.2</td>
<td>CS</td>
<td>Sophomore</td>
</tr>
<tr>
<td>2222</td>
<td>09/01/2014</td>
<td>3.8</td>
<td>Null</td>
<td>Freshman</td>
</tr>
</tbody>
</table>

FreshRel

<table>
<thead>
<tr>
<th>Id</th>
<th>SD</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2222</td>
<td>09/01/2014</td>
<td>3.8</td>
</tr>
</tbody>
</table>

and SophoRel,

<table>
<thead>
<tr>
<th>Id</th>
<th>SD</th>
<th>GPA</th>
<th>Major</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>09/01/2013</td>
<td>3.2</td>
<td>CS</td>
</tr>
</tbody>
</table>

Both FreshRel and SophoRel will be implemented as a dynamic view of the StudSum relation, thus will be automatically revised when a new student is entered there.
A summary

Below shows the conversion from the IsA structure to a table structure for the example.

PersonSum(SSN,N,DoB,Dt,Sal,Sp,Cat,SD,GPA,Major,
         Advisor,Status,Cat,Kind); --Disjoint
EmplSum(SSN,Name,DoB,Dt,Sal,Sp,Cat);--Disjoint
   FK: SSN references PersonSum(SSN);--Participation
EmplRel(SSN,Name,DoB,Dt,Sal);--Coverage
   FK: SSN references EmplSum(SSN);--Participation
SecRel(SSN,Name,DoB,Dt,Sal); --Coverage
   FK: SSN references EmplSum(SSN);--Participation
TechRel(SSN,Name,DoB,Dt,Sal,Sp); --Coverage
   FK: SSN references EmplSum(SSN);--Participation

StudSum(SSN,Name,DoB,SD,GPA,Major,Advisor,
         Status);--Disjoint
StudRel(SSN,Name,DoB,SD,GPA);--Coverage
   FK: SSN references StudSum(SSN);--Participation
FreshRel(SSN,Name,DoB,SD,GPA);--Coverage
   FK: SSN references StudSum(SSN);--Participation
SophRel(SSN,Name,DoB,SD,GPA,Major);--Coverage
   FK: SSN references StudSum(SSN);--Participation
JrRel(SSN,Name,DoB,SD,GPA,Major);--Coverage
   FK: SSN references StudSum(SSN);--Participation
SrRel(SSN,Name,DoB,SD,GPA,Major,Advisor);--Coverage
   FK: SSN references StudSum(SSN);--Participation
Represent participation constraint

It is pretty easy to represent the fact that every instance of one type is included in an instance of another type, with a referential integrity constraint.

For example, every professor works in at least one department, we simply add in a foreign key clause.

Create table Professor (  
    Id    Integer,  
    Name  Char(20),  
    Primary key (Id),  
    Foreign key (Id)  
    references WorksIn(ProfId))

In this case, a relationship is implemented by a foreign key constraint.
Another possibility

We can also merge the WorksIn relationship with the Professor entity and derive the following table:

Create table ProfessorWithWorksIn (  
    Id Integer,  
    Name Char(20),  
    DeptId Char(4) Not Null,  
    --Enforcing the participation --constraint  
    Since Date,  
    Primary key (Id),  
    Foreign key DeptId  
    references Department)

**Question:** Which one to use?

**Answer:** It is a subjective decision ....

**Homework:** 4.4.
Another example

Participation can not always be done via a foreign key. For example, we can have the following for the transcript relation.

Create table Transcript ( 
    StudId Integer, 
    CrsCode Char(6), 
    Semester Char(6), 
    Grade Char(1), 
    Primary key (StudId, CrsCode, Semester), 
    Foreign key (StudId) 
        references Student(Id), 
    Foreign key (CrsCode) 
        references Course(CrsCode), 
    Foreign key (Semester) 
        references Semester (SemCode))
What else?

We have to also say that every student has to take at least one course, by requiring that every student Id occurs at least once in the Transcript table.

But, since StudId is not a primary key of the Transcript table, it is not a foreign key. We thus have to include the following in the Student table.

Student(ID) references Transcript (StudId)
Non-exclusive part-of

If an entity does not need to be a part of a relationship *exclusively*, we simply treat it as a regular one with nothing special.

Moreover, if this entity can participate in at most one relation, we can simply merge the entity and the relationship together, as we did with the Professor and WorksIn types.

A foreign key in the merged table will refer to the original entity that contains the subpart.

Since this entity does not need to be part of this relation, for those missing parts, we will put in `null` values for some of the attributes in the merged table.

In particular, we will not require `not null` for this involved foreign key.
An example

Given the following diagram for automobile and wheel, we merge the two entities

Since a wheel need not be part of a vehicle, we can merge Wheel and Part-of as follows:

Create table WheelWithPartOf (  
  SerialNumber Integer,  
  Size Char(20),  
  Manufacturer Char(20),  
  VehicleId Char(20), --Where is the tire?  
  Primary key (SerialNumber),  
  Foreign key (VehicleId)  
    references Automobile)

Notice that VehicleId does not come with a not null clause.

Question: Why the specific constraint?
For the exclusive ones

In this case, the subtype is a weak entity, and the partOf relationship is its identifying relationship. Thus, the associated foreign key must come with the not null clause, since its key must exist somewhere else, and when that goes, so does this weak entity instance.

For example, Professor is a weak entity, depending on a department.

Create table ProfessorWithWorksIn (  
  Id Integer,  
  Name Char(20),  
  DeptId Char(4) Not Null,  
    --Enforcing the participation  
    --constraint  
  Since Date,  
  Primary key (Id),  
  Foreign key DeptId  
    references Department)
Limitations

We have seen that data modeling methodologies such as the E/R modeling techniques are very useful to get us started, although it takes lots of efforts, experience, and intuition to get them well done, but not too tender.

It is not always this easy.... There are certain issues that are not completely resolved.

Let's consider a few....
Entity or attribute?

In the following design, we represent Semester as an entity with SemCode as its key, which leads to a ternary implementation for the Transcript entity type.

Another possibility is to include SemCode into the Transcript type so that the latter relation contains two roles, Student and Course; together with two attributes, Grade and SemCode.

**Review:** What does that thick line mean?
It really depends,...

In many cases, whether to represent some object as a separate entity or an attribute of another entity is a personal choice.

But, the real issue is whether this object has its own structure. If it does, it should be put into an entity by itself.

In this case, if Semester has nothing but a SemCode, it would be an overkill to implement it as an entity. On the other hand, if it has other stuff, such as StartDate, EndDate, Holidays, etc, then it is justified to represent it as an entity.
Entity or relationship?

Notice that in this case, Transcript can also be represented as an entity and another relationship Enrolled can be brought in to connect Transcript, Student and Course: a student is enrolled in a course and is given a grade.

We might want to minimize the total number of entities and relationships since this number has a direct impact on the number of tables.

It is not a biggie if we lump a few relationships into one big type since later on, under the guidance of the normalization theory, we can always decompose them appropriately.

It is however much more challenging to combine the pieces later on.
More or less?

Sometimes, when we change the degree of a relationship, information will get lost as a result. For example, in the following diagram, people might want to replace ternary relationships with a bunch of binary ones.

As a result, they might come up with the following:
What’s wrong?

1. It introduces some gap among entities. For example, under the original ternary relationship Sold, there might be an instance saying that a supplier, ‘Ace’, might have sold some ‘screws’ to a project ‘P1’, which implies that ’P1’ uses those screws supplied by ’Ace’.

With the new design, ’Ace’ sells screws, and project ’P1’ uses screws. But, we no long have the information if ’P1’ uses screws supplied by ’Ace’. 😞

Thus, certain information has got lost.
What else?

2. Another issue is that Price is associated with the Supplies relationship, where customer does not play a role. This makes the whole thing less flexible since now price is fixed for all the projects, while in the original design, it might depend on the business partners, and perhaps the associated quantity.

3. With the new design, the supplier can do only one business with a project every day 😞, since we cannot have the same pair of (Supplier, Project) with different dates, as this pair of attributes might serve as the primary key.

On the other hand, with the original one, it is doable as long as we sell different parts.
Should we always go higher?

In the following diagram, *the bad guy*,

**Question:** What does that thick arrow mean?

**Question:** Can we combine HasAccount, WorksIn and IsHandledBy into one with much higher degree?

Definitely a Shakespearean moment... 😊
... to keep or not to keep?

If we keep the thick arrow from Broker to WorksIn, i.e., “Every broker works in exactly one office.” we will then introduce the constraints saying that every broker can have at most one account and at most one client, since, with Broker being the key, we don’t allow two distinct rows in such a relation, sharing the same broker number.

On the other hand, if we drop such a connection, we will then drop the requirement that every broker will be assigned to only one office. Some one can work in both New York and Boston, which might not be desirable, either.

You should be ready to do Project (II).

Homework: 4.7(a, b).