Chapter 13
Semantic Models

Database systems typically have only a very limited understanding of what the data in the database means: certain simple data values, and perhaps certain simple constraints that apply to those values, and very little else.

It would be great if, e.g., SQL understands that part weights and shipment quantities, though both of numerical values, are semantically different in their nature, so that it will at least question a request which tries to join parts and shipments on matching weights and quantities.

We will try to study some of the ideas behind the activities to add in more semantical features into database models. More importantly, we will see how we could build up a “rough” database, based on those semantic features.
The overall approach

1. Set up a data model: We try to identify a set of semantic concepts that seem to be useful in telling informally about the real world. E.g., The world is made of entities, which can be classified into entity types. Entities of the same type might share certain properties, one of them might provide an identity for various entities. Then, entity can be related to other entities by means of relationship.

2. Collect the tables: We then try to devise a set of corresponding symbolic objects that can be used to represent the aforementioned semantic concepts. For example, we can use the table S to collect all the information for the suppliers.
3. **Establish the FD set:** We also devise a set of formal *integrity rules* to enforce certain relationship among those formal objects. For example, every table has a unique kep attribute.

4. **Design the queries:** Finally, we also develop a set of formal *operators* for manipulating those formal objects. For example, `Select SNAME from S where CITY='London';`
The E/R model

*Entity/relationship (E/R)* model is one of the best-known semantic modeling approaches. It includes analogs of all of the objects mentioned before such as entities, properties, relationships, and types. It also comes up with some *diagramming technique*. Below shows such an example:
Entities

*Entity* is defined as *a thing which can be distinctly identified*. More specifically, a *weak entity* is an entity that is existence-dependent on some other entity, in the sense that it can’t exist if that other entity does not also exist.

For example, An employee’s dependents might be weak entities. thus, if an employee is deleted from the database, all his/her dependents must be deleted, too. A *regular entity* is an entity that is not weak, e.g., an employee might be regular entities.

**Question:** How can we represent the concept of *weak entity*?

**Answer:** Foreign key.
Properties of an entity

Entities, and relationships, have properties. All entities or a relationship of a given type have certain properties in common. e.g., all employees has a supervisor.

Properties can be simple or composite; single or multi-valued; missing (unknown); base or derived, e.g., “total quantity” can be derived as the sum of the individual shipment quantities for that part.

**Question:** How can we represent the concept of entity and properties?

**Answer:** Tables and attributes.
Relationship

*Relationship* is defined as *an association among entities*. E.g., there is a relationship `DEPT_EMP` between departments and employees, representing the fact that certain departments employ certain employees. The entities involved in a relationship are called *participants* of that relationship. The number of such participants of a relationship is called its degree.

Let $R$ have entity type $E$ as a participant. If every instance of $E$ participates in at least one instance of $R$, then the participation of $E$ in $R$ is *total*, otherwise, it is *partial*. E.g., if every employee must belong to a department, then the participation of employees in `DEPT_EMP` is total.
The nature of a relationship

A relationship between two entities can be one-to-one, one-to-many, or many-to-many.

For example, the relationship *teaching* between the instructor entity and the course entity is one-to-many, since one instructor can teach several courses, but given the CRN number, there is only one instructor. Similarly, the relationship *publishedBy* between the book entity and the publisher entity is also a one-to-many relation.

But, the relationship *shipment* between the supplier entity and the part entity is many-to-many. Similarly, the *checkOut* relationship between the reader entity and the book entity is also a many-to-many.

The one-to-one is a special one of the one-to-many class.
Represent a M-M relationship

Given a many-to-many relationship between two entities, e.g., the shipment relation between S and P, besides using two base tables to represent the two entities, together with their properties, we use another table, in this case SP, to represent this relationship and its properties, QTY in this case. This leads to S, P, and SP.

The reason is that, in this case, the property of this relation, QTY functionally depends on both S# and P#, thus if we include QTY in any of the two tables, we will have a table that is not in the 2NF.

In fact, even if the relationship does not have any property, we still have to have a separate table to represent this relationship to have a 2NF table.
Represent a 1-M relationship

Given a one-to-many relationship between two entities, e.g., the teaching relation between I(Instructor\{I\#,\ldots\}) and C(Course\{CRN,\ldots\}), we use two base tables to represent the two entities, then add in the I\# into the associated Course rows, and let this I\# be a foreign key referring back to the I table. In other words, we add in the information from the one side to the M side.

The reason is that, in this case, the 1-M relation between I\# and CRN is really a FD: CRN $\rightarrow$ I\#, and we notice in this case, for every attribute A in the C table, we have CRN $\rightarrow$ A. Thus, the one-side is essentially a property of the M-side.
On the other hand, if we put the CRN into the I table instead, then since for every attribute B in the I table, \( I\# \rightarrow B \), we have \( CRN \rightarrow I\# \rightarrow B \) by transitivity. Notice in this case, the table contains redundancy, since CRN, a nonkey attribute, does not depend on I\#, the primary key.

We also notice that in this case, any property associated with such a relation is essentially a property of the M-side. For example, the enrollment of some one teaching the course is essentially a property of the courses, but not that of the teaching relationship.
Represent a 1-1 relationship

Given a one-to-one relationship between two entities, for example, the managing relationship between Emp(Employee{Emp#, ...}) and B(Branch{B#, ...}), we use two base tables to represent the two entities, then either add in the B# into the associated Emp table rows; or add in the Emp# into the associated B table rows. We also add in a foreign key constraint in both cases.

Notice that in this case, we have to FDs: Emp# → B# and B# → Emp#. Thus, we can either regard Emp# to be a property of B#, or vice versa.
Among types

Any entity is of at least one entity type, but an entity can be of several types at the same time. For example, if some employees are programmers, then we might say that entity type PROGRAMMER is a subtype of EMPLOYEE.

Similarly, we can have the concept of a supertype. The neat thing (?) is that then all properties of employees apply automatically to programmers, but the converse is not true. E.g., programmers might have a property “Java skill”, which does not apply to employees in general. Likewise, programmers automatically participate in all relationships in which employees participate, but the converse is not true. E.g., programmers might belong to IEEE, while employees in general do not.
Note further that there could be more sub(supers) types, e.g., APPLICATION_PROGRAMMER and SYSTEM_PROGRAMMER, etc.. Thus, in general, we can have an entity type hierarchy. For example,

![Entity Type Hierarchy Diagram]

**Question:** Did we see this stuff before?

**Answer:** Property inheritance in OOP.
E/R diagrams

E/R diagrams is simply a technique for representing the logical structure of a database in a pictorial manner. They provide a simple means of communicating the salient features of the design of any given database.

Each entity is shown as a *rectangle* containing the name of the entity type. A weak entity type has a double border.

Properties are shown as *ellipses* containing the name of the property, and attached to the relevant entity or relationship via a solid line. Again, its border can be dotted or dashed, depending on if the property is derived or multi-valued. Key properties are *underlined*. 
Each relationship is shown as a diamond containing its name. The border is doubled if the relationship is between a weak entity and the entity type on which it depends. The participants are connected to the relationship with a solid line; and each line is labeled “1” or “M” to indicate if the relationship is 1-1, or 1-many, etc.

If entity \( Y \) is a subtype of \( X \), we draw a solid line from \( X \) to \( Y \), marked with an arrowhead at the \( Y \) end. This line denotes the \( isa \) relationship, since each instance of \( Y \) is also a \( X \) instance.
Design with the E/R model

It should be clear that an E/R diagram as described is a (first stage) database design. But, it is still very imprecise, if we want to map it into a specific DBMS. We will discuss how to do it via an example.

We have collected the following regular entities from the diagram: DEPARTMENT, EMPLOYEE, SUPPLIER, PART, PROJECT. Each of them maps into a base relvar. The database thus contains five entity types: DEPT, EMP, S, P, and J. Each of them will have a candidate key, as indicated in the diagram, which is used as their respective primary key. E.g., the definition of the DEPT relvar can be the following:

```
VAR DEPT BASE RELATION
   {DEPT#, ..., ...} 
PRIMARY KEY {DEPT#};
```
Many-to-many relationships

We have several many-to-many relationships, e.g., SUPP_PART, which involves suppliers and parts. Let the relvar for the relationship be SP, i.e.,

VAR SP BASE RELATION
{S#, ..., P#, ...}
PRIMARY KEY {S#, P#}
FOREIGN KEY {S#} REFERENCES S
  ON DELETE RESTRICT
  ON UPDATE CASCADE
FOREIGN KEY {P#} REFERENCES P
  ON DELETE RESTRICT
  ON UPDATE CASCADE;
Many-to-one relationships

We also have several many-to-one relationships, e.g., DEPT_EMP. We will not introduce any new relvar, but simply introduce a foreign key in the relvar on the “many” side (EMP) that references the relvar on the “one” side (DEPT).

VAR EMP BASE RELATION
{EMP#, ..., DEPT#, ...}
PRIMARY KEY {EMP#}
FOREIGN KEY {DEPT#} REFERENCES DEPT
  ON DELETE ...
  ON UPDATE ...;
Weak entities

The relationship from a weak entity type to the entity type on which it depends is naturally a many-to-one relationship. E.g., the relationship from DEPENDENTS to EMP is such an relationship.

VAR DEPENDENT BASE RELATION
{EMP#,...,...,...}
PRIMARY KEY{EMP#, DEP_NAME}
FOREIGN KEY {EMP#} REFERENCES EMP
ON DELETE CASCADE
ON UPDATE CASCADE

Each property shown in the diagram maps into an attribute in an appropriate relvar.
Represent the hierarchy

If we consider the types EMPLOYEE and PROGRAMMER, then the supertype EMPLOYEE maps into a base relvar, EMP, and the subtype PROGRAMMER, containing properties possessed by programmes only, maps into another base relvar, PGMR, with primary key the same as the supertype relvar and with other attributes corresponding to the properties that apply only to employees who are programmer, e.g., LANG. Furthermore, PGMR primary key is also a foreign key, referring back to the EMP relvar. Thus,

VAR PGMR BASE RELATION
   {EMP#, ..., LANG, ...}
PRIMARY KEY {EMP#}
FOREIGN KEY {EMP#} REFERENCES EMP
   ON DELETE CASCADE
   ON UPDATE CASCADE;
We also need the following view, EMP_PGMR,

VAR EMP_PGMR VIEW
  EMP JOIN PGMR;

This view contains just those employees who are programmers.

With this design, we can access properties that apply to all the employees by using EMP; access to those only applied to programmer using PGMR; and access to all programmer related properties of employees by using the view EMP_PGMR; we can insert employees who are not programmers by using EMP, and insert programmers by using PGMR; we can make an existing non programmer into a programmer by inserting the employee into either base relvar PGMR, or view EMP_PGMR; we can make an existing programmer into a non programmer by deleting the programmer from base relvar PGMR.
Homework assignment

1. Read through §14.6.

2. Try exercises 14.5 and 14.7.