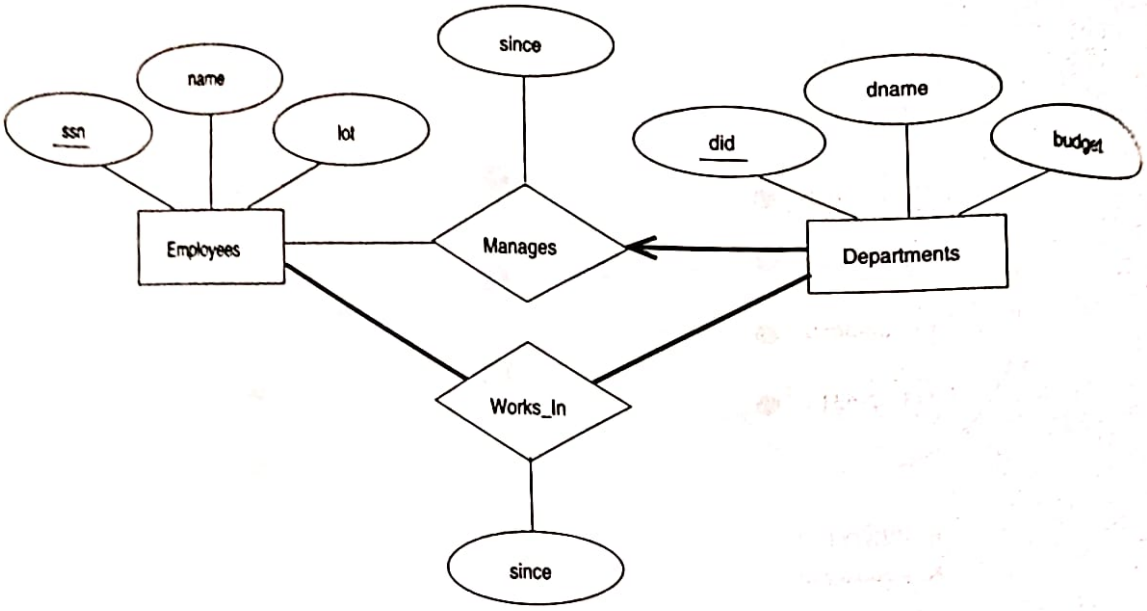

Figure 2.10 Manages and Works_In

ssn; thus we might have two employees called Smethurst, and each might have a son called Joe.

Dependents is an example of a **weak entity set**. A weak entity can be identified uniquely only by considering some of its attributes in conjunction with the primary key of another entity, which is called the **identifying owner**.

The following restrictions must hold:

- The owner entity set and the weak entity set must participate in a one-to-many relationship set (one owner entity is associated with one or more weak entities, but each weak entity has a single owner). This relationship set is called the **identifying relationship set** of the weak entity set.
- The weak entity set must have total participation in the identifying relationship set.

For example, a Dependents entity can be identified uniquely only if we take the key of the *owning* Employees entity and the *pname* of the Dependents entity. The set of attributes of a weak entity set that uniquely identify a weak entity for a given owner entity is called a *partial key* of the weak entity set. In our example ***pname*** is a partial key for Dependents.

The Dependents weak entity set and its relationship to Employees is shown in Figure 2.11. The total participation of Dependents in Policy is indicated by linking them

with a dark line. The arrow from Dependents to Policy indicates that each Dependents entity appears in at most one (indeed, exactly one, because of the participation constraint) Policy relationship. To underscore the fact that Dependents is a weak entity and Policy is its identifying relationship, we draw both with dark lines. To indicate that *pname* is a partial key for Dependents, we underline it using a broken line. This means that there may well be two dependents with the same *pname* value.

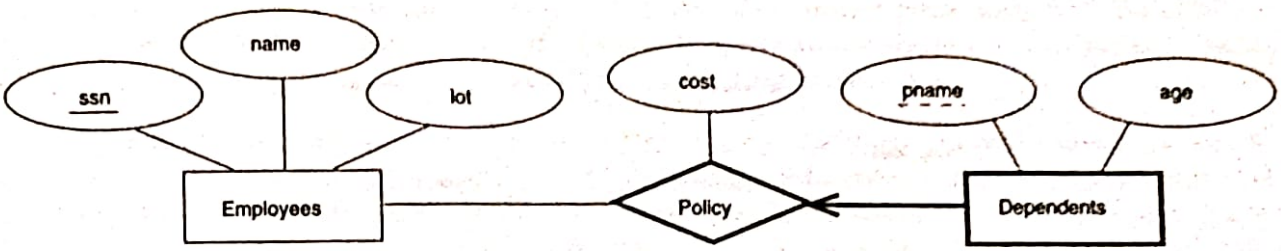

Figure 2.11 A Weak Entity Set

2.4.4 Class Hierarchies

Sometimes it is natural to classify the entities in an entity set into subclasses. For example, we might want to talk about an Hourly_Emps entity set and a Contract_Emps entity set to distinguish the basis on which they are paid. We might have attributes *hours_worked* and *hourly_wage* defined for Hourly_Emps and an attribute *contractid* defined for Contract_Emps.

We want the semantics that every entity in one of these sets is also an Employees entity, and as such must have all of the attributes of Employees defined. Thus, the attributes defined for an Hourly_Emps entity are the attributes for Employees plus Hourly_Emps. We say that the attributes for the entity set Employees are **inherited** by the entity set Hourly_Emps, and that Hourly_Emps **ISA** (read *is a*) Employees. In addition—and in contrast to class hierarchies in programming languages such as C++—there is a constraint on queries over instances of these entity sets: A query that asks for all Employees entities must consider all Hourly_Emps and Contract_Emps entities as well. Figure 2.12 illustrates the class hierarchy.

The entity set Employees may also be classified using a different criterion. For example, we might identify a subset of employees as Senior_Emps. We can modify Figure 2.12 to reflect this change by adding a second ISA node as a child of Employees and making Senior_Emps a child of this node. Each of these entity sets might be classified further, creating a multilevel ISA hierarchy.

A class hierarchy can be viewed in one of two ways:

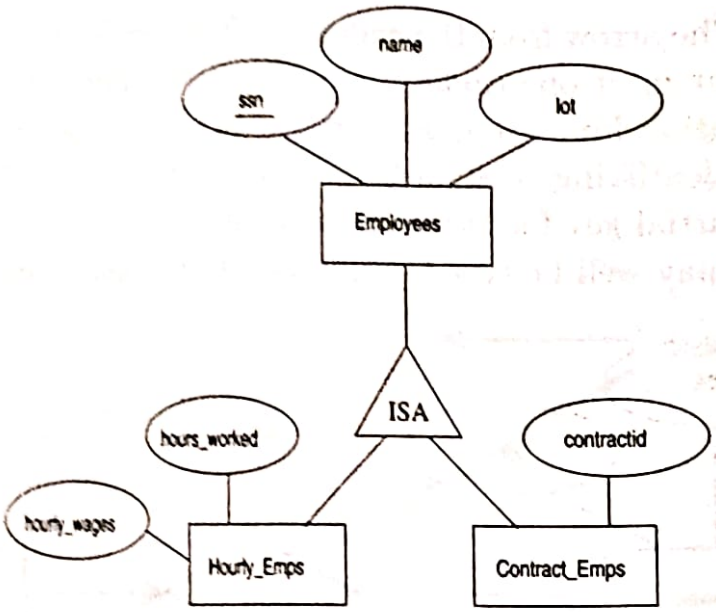

Figure 2.12 Class Hierarchy

- Employees is **specialized** into subclasses. Specialization is the process of identifying subsets of an entity set (the **superclass**) that share some distinguishing characteristic. Typically the superclass is defined first, the subclasses are defined next, and subclass-specific attributes and relationship sets are then added.

- Hourly_Emps and Contract_Emps are **generalized** by Employees. As another example, two entity sets Motorboats and Cars may be generalized into an entity set Motor_Vehicles. Generalization consists of identifying some common characteristics of a collection of entity sets and creating a new entity set that contains entities possessing these common characteristics. Typically the subclasses are defined first, the superclass is defined next, and any relationship sets that involve the superclass are then defined.

We can specify two kinds of constraints with respect to ISA hierarchies, namely, *overlap* and *covering* constraints. **Overlap constraints** determine whether two subclasses are allowed to contain the same entity. For example, can Attishoo be both an Hourly_Emps entity and a Contract_Emps entity? Intuitively, no. Can he be both a Contract_Emps entity and a Senior_Emps entity? Intuitively, yes. We denote this by writing 'Contract_Emps OVERLAPS Senior_Emps.' In the absence of such a statement, we assume by default that entity sets are constrained to have no overlap.

Covering constraints determine whether the entities in the subclasses collectively

Motor_Vehicles.' In the absence of such a statement, we assume by default that there is no covering constraint; we can have motor vehicles that are not motorboats or cars.

There are two basic reasons for identifying subclasses (by specialization or generalization):

1. We might want to add descriptive attributes that make sense only for the entities in a subclass. For example, *hourly_wages* does not make sense for a Contract_Emps entity, whose pay is determined by an individual contract.
2. We might want to identify the set of entities that participate in some relationship. For example, we might wish to define the Manages relationship so that the participating entity sets are Senior_Emps and Departments, to ensure that only senior employees can be managers. As another example, Motorboats and Cars may have different descriptive attributes (say, tonnage and number of doors), but as Motor_Vehicles entities, they must be licensed. The licensing information can be captured by a Licensed_To relationship between Motor_Vehicles and an entity set called Owners.

2.4.5 Aggregation

As we have defined it thus far, a relationship set is an association between entity sets. Sometimes we have to model a relationship between a collection of entities and *relationships*. Suppose that we have an entity set called Projects and that each Projects entity is sponsored by one or more departments. The Sponsors relationship set captures this information. A department that sponsors a project might assign employees to monitor the sponsorship. Intuitively, Monitors should be a relationship set that associates a Sponsors relationship (rather than a Projects or Departments entity) with an Employees entity. However, we have defined relationships to associate two or more *entities*.

In order to define a relationship set such as Monitors, we introduce a new feature of the ER model, called *aggregation*. **Aggregation** allows us to indicate that a relationship set (identified through a dashed box) participates in another relationship set. This is illustrated in Figure 2.13, with a dashed box around Sponsors (and its participating entity sets) used to denote aggregation. This effectively allows us to treat Sponsors as an entity set for purposes of defining the Monitors relationship set.

When should we use aggregation? Intuitively, we use it when we need to express a relationship among relationships. But can't we express relationships involving other relationships without using aggregation? In our example, why not make Sponsors a ternary relationship? The answer is that there are really two distinct relationships, Sponsors and Monitors, each possibly with attributes of its own. For instance, the

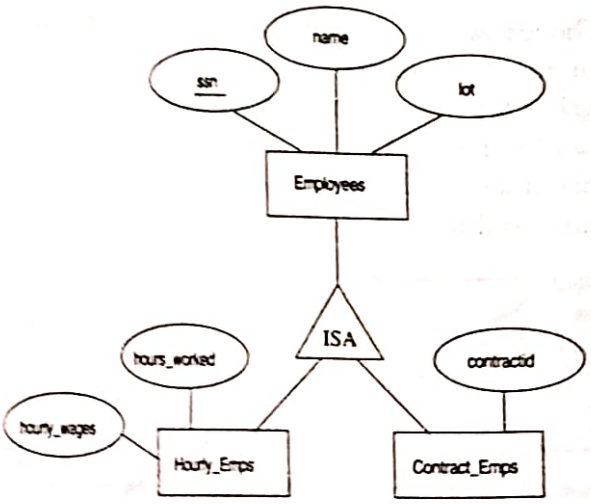

Figure 2.12 Class Hierarchy

- Employees is **specialized** into subclasses. Specialization is the process of identifying subsets of an entity set (the **superclass**) that share some distinguishing characteristic. Typically the superclass is defined first, the subclasses are defined next, and subclass-specific attributes and relationship sets are then added.
- Hourly_Emps and Contract_Emps are **generalized** by Employees. As another example, two entity sets Motorboats and Cars may be generalized into an entity set Motor_Vehicles. Generalization consists of identifying some common characteristics of a collection of entity sets and creating a new entity set that contains entities possessing these common characteristics. Typically the subclasses are defined first, the superclass is defined next, and any relationship sets that involve the superclass are then defined.

We can specify two kinds of constraints with respect to ISA hierarchies, namely, *overlap* and *covering* constraints. **Overlap constraints** determine whether two subclasses are allowed to contain the same entity. For example, can Attishoo be both an Hourly_Emps entity and a Contract_Emps entity? Intuitively, no. Can he be both a Contract_Emps entity and a Senior_Emps entity? Intuitively, yes. We denote this by writing 'Contract_Emps OVERLAPS Senior_Emps.' In the absence of such a statement, we assume by default that entity sets are constrained to have no overlap.

Covering constraints determine whether the entities in the subclasses collectively include all entities in the superclass. For example, does every Employees entity have to belong to one of its subclasses? Intuitively, no. Does every Motor_Vehicles entity have to be either a Motorboats entity or a Cars entity? Intuitively, yes; a characteristic property of generalization hierarchies is that every instance of a superclass is an instance of a subclass. We denote this by writing 'Motorboats AND Cars COVER

Motor_Vehicles.' In the absence of such a statement, we assume by default that there is no covering constraint; we can have motor vehicles that are not motorboats or cars.

There are two basic reasons for identifying subclasses (by specialization or generalization):

1. We might want to add descriptive attributes that make sense only for the entities in a subclass. For example, *hourly_wages* does not make sense for a Contract_Emps entity, whose pay is determined by an individual contract.

2. We might want to identify the set of entities that participate in some relationship. For example, we might wish to define the Manages relationship so that the participating entity sets are Senior_Emps and Departments, to ensure that only senior employees can be managers. As another example, Motorboats and Cars may have different descriptive attributes (say, tonnage and number of doors), but as Motor_Vehicles entities, they must be licensed. The licensing information can be captured by a Licensed_To relationship between Motor_Vehicles and an entity set called Owners.

2.4.5 Aggregation

As we have defined it thus far, a relationship set is an association between entity sets. Sometimes we have to model a relationship between a collection of entities and *relationships*. Suppose that we have an entity set called Projects and that each Projects entity is sponsored by one or more departments. The Sponsors relationship set captures this information. A department that sponsors a project might assign employees to monitor the sponsorship. Intuitively, Monitors should be a relationship set that associates a Sponsors relationship (rather than a Projects or Departments entity) with an Employees entity. However, we have defined relationships to associate two or more *entities*.

In order to define a relationship set such as Monitors, we introduce a new feature of the ER model, called *aggregation*. **Aggregation** allows us to indicate that a relationship set (identified through a dashed box) participates in another relationship set. This is illustrated in Figure 2.13, with a dashed box around Sponsors (and its participating entity sets) used to denote aggregation. This effectively allows us to treat Sponsors as an entity set for purposes of defining the Monitors relationship set.

When should we use aggregation? Intuitively, we use it when we need to express a relationship among relationships. But can't we express relationships involving other relationships without using aggregation? In our example, why not make Sponsors a ternary relationship? The answer is that there are really two distinct relationships, Sponsors and Monitors, each possibly with attributes of its own. For instance, the

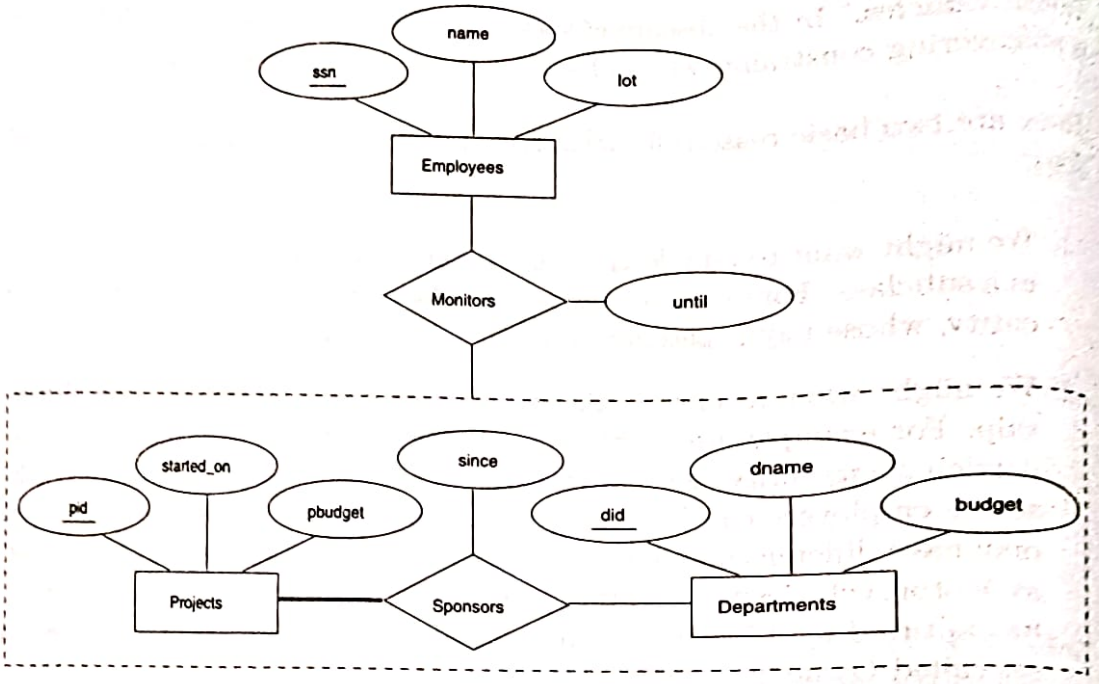

Figure 2.13 Aggregation

Monitors relationship has an attribute *until* that records the date until when the employee is appointed as the sponsorship monitor. Compare this attribute with the attribute *since* of Sponsors, which is the date when the sponsorship took effect. The use of aggregation versus a ternary relationship may also be guided by certain integrity constraints, as explained in Section 2.5.4.

2.5 CONCEPTUAL DATABASE DESIGN WITH THE ER MODEL

Developing an ER diagram presents several choices, including the following:

- Should a concept be modeled as an entity or an attribute?
- Should a concept be modeled as an entity or a relationship?
- What are the relationship sets and their participating entity sets? Should we use binary or ternary relationships?
- Should we use aggregation?

We now discuss the issues involved in making these choices.

2.5.1 Entity versus Attribute

While identifying the attributes of an entity set, it is sometimes not clear whether a property should be modeled as an attribute or as an entity set (and related to the first entity set using a relationship set). For example, consider adding address information to the Employees entity set. One option is to use an attribute *address*. This option is appropriate if we need to record only one address per employee, and it suffices to think of an address as a string. An alternative is to create an entity set called Addresses and to record associations between employees and addresses using a relationship (say, Has_Address). This more complex alternative is necessary in two situations:

- We have to record more than one address for an employee.
- We want to capture the structure of an address in our ER diagram. For example, we might break down an address into city, state, country, and Zip code, in addition to a string for street information. By representing an address as an entity with these attributes, we can support queries such as "Find all employees with an address in Madison, WI."

For another example of when to model a concept as an entity set rather than as an attribute, consider the relationship set (called Works_In2) shown in Figure 2.14.

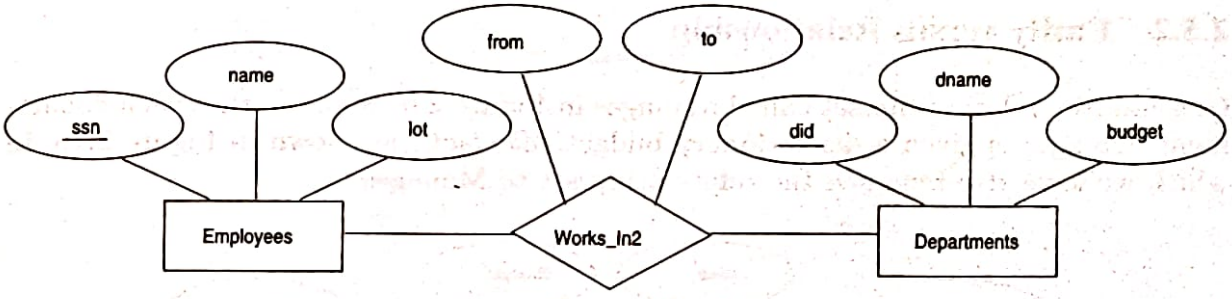

Figure 2.14 The Works_In2 Relationship Set

It differs from the Works_In relationship set of Figure 2.2 only in that it has attributes *from* and *to*, instead of *since*. Intuitively, it records the interval during which an employee works for a department. Now suppose that it is possible for an employee to work in a given department over more than one period.

This possibility is ruled out by the ER diagram's semantics. The problem is that we want to record several values for the descriptive attributes for each instance of the Works_In2 relationship. (This situation is analogous to wanting to record several addresses for each employee.) We can address this problem by introducing an entity set called, say, Duration, with attributes *from* and *to*, as shown in Figure 2.15.

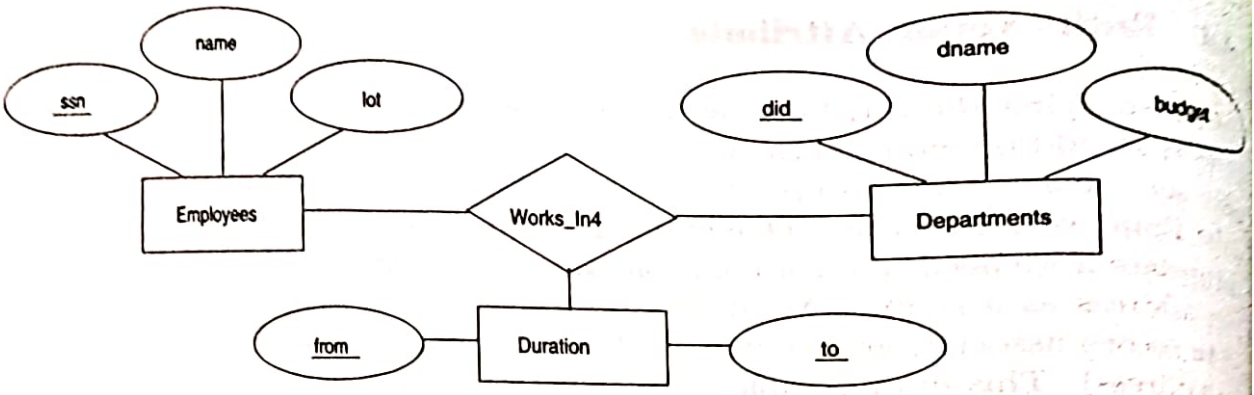

Figure 2.15 The Works_In4 Relationship Set

In some versions of the ER model, attributes are allowed to take on sets as values. Given this feature, we could make Duration an attribute of Works_In, rather than an entity set; associated with each Works_In relationship, we would have a set of intervals. This approach is perhaps more intuitive than modeling Duration as an entity set. Nonetheless, when such set-valued attributes are translated into the relational model, which does not support set-valued attributes, the resulting relational schema is very similar to what we get by regarding Duration as an entity set.

2.5.2 Entity versus Relationship

Consider the relationship set called Manages in Figure 2.6. Suppose that each department manager is given a discretionary budget (*dbudget*), as shown in Figure 2.16, which we have also renamed the relationship set to Manages2.

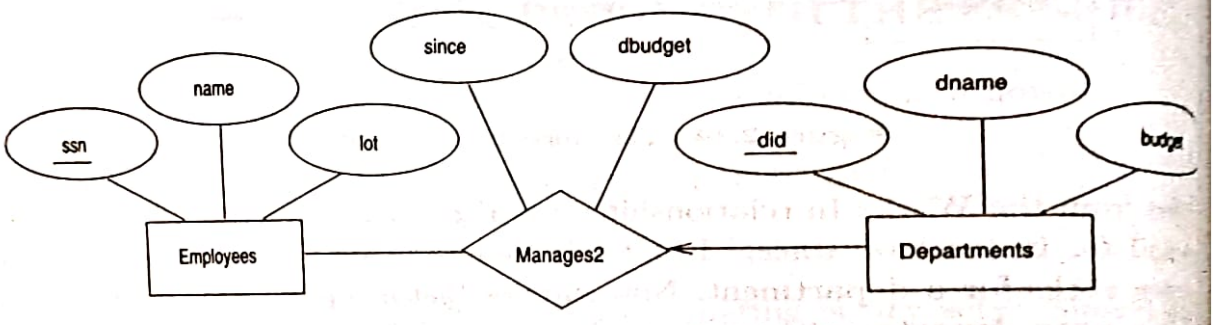

Figure 2.16 Entity versus Relationship

There is at most one employee managing a department, but a given employee can manage several departments; we store the starting date and discretionary budget for each manager-department pair. This approach is natural if we assume that a manager receives a separate discretionary budget for each department that he or she manages

But what if the discretionary budget is a sum that covers *all* departments managed by that employee? In this case each Manages2 relationship that involves a given employee will have the same value in the *dbudget* field. In general such redundancy could be significant and could cause a variety of problems. (We discuss redundancy and its attendant problems in Chapter 15.) Another problem with this design is that it is misleading.

We can address these problems by associating *dbudget* with the appointment of the employee as manager of a *group* of departments. In this approach, we model the appointment as an entity set, say Mgr_Appt, and use a ternary relationship, say Manages3, to relate a manager, an appointment, and a department. The details of an appointment (such as the discretionary budget) are not repeated for each department that is included in the appointment now, although there is still one Manages3 relationship instance per such department. Further, note that each department has at most one manager, as before, because of the key constraint. This approach is illustrated in Figure 2.17.

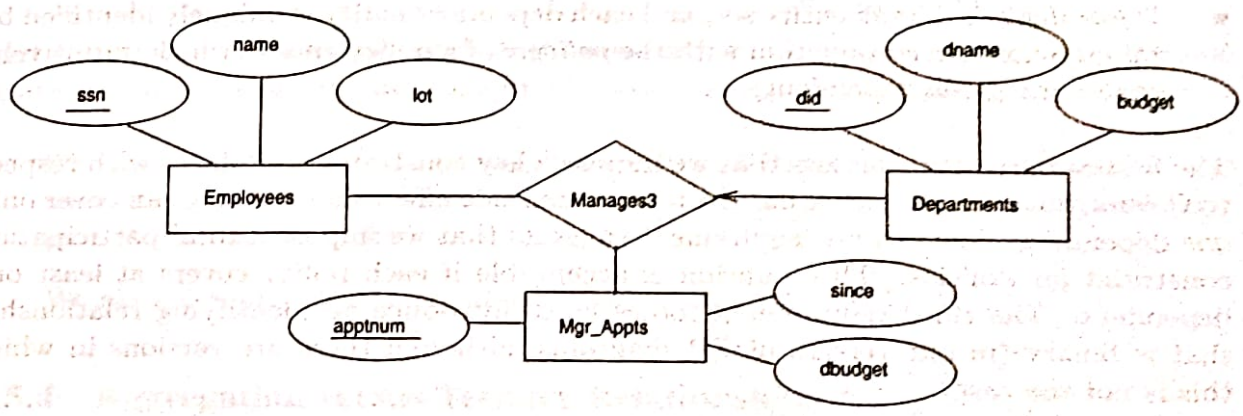

Figure 2.17 Entity Set versus Relationship

2.5.3 Binary versus Ternary Relationships *

Consider the ER diagram shown in Figure 2.18. It models a situation in which an employee can own several policies, each policy can be owned by several employees, and each dependent can be covered by several policies.

Suppose that we have the following additional requirements:

- A policy cannot be owned jointly by two or more employees.
- Every policy must be owned by some employee.

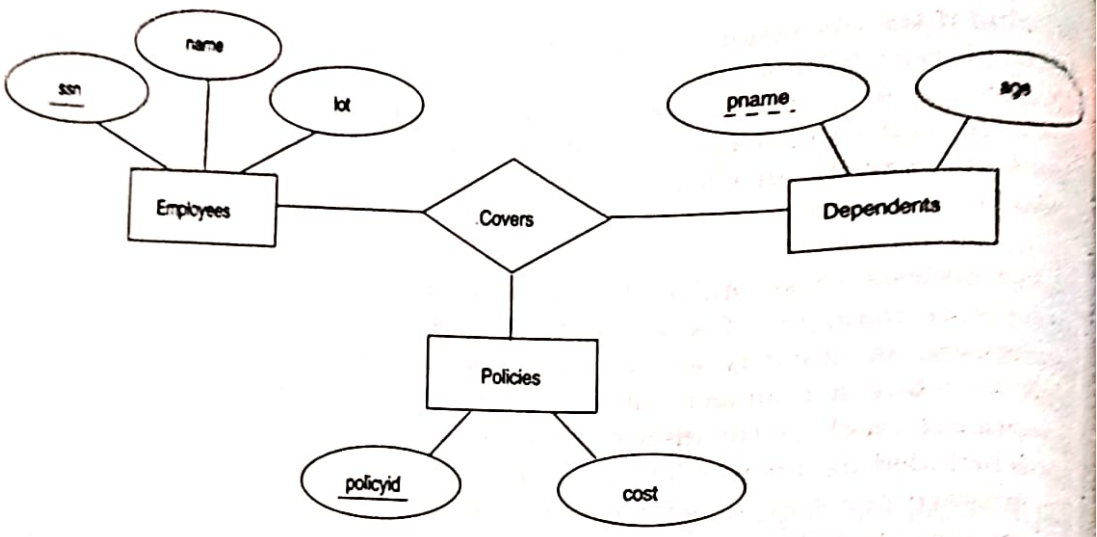

Figure 2.18 Policies as an Entity Set

- Dependents is a weak entity set, and each dependent entity is uniquely identified by taking *pname* in conjunction with the *policyid* of a policy entity (which, intuitively, covers the given dependent).

The first requirement suggests that we impose a key constraint on Policies with respect to Covers, but this constraint has the unintended side effect that a policy can cover only one dependent. The second requirement suggests that we impose a total participation constraint on Policies. This solution is acceptable if each policy covers at least one dependent. The third requirement forces us to introduce an identifying relationship that is binary (in our version of ER diagrams, although there are versions in which this is not the case).

Even ignoring the third point above, the best way to model this situation is to use two binary relationships, as shown in Figure 2.19.

2.4.2

This example really had two relationships involving Policies, and our attempt to model a single ternary relationship (Figure 2.18) was inappropriate. There are situations, however, where a relationship inherently associates more than two entities. We have seen such an example in Figure 2.4 and also Figures 2.15 and 2.17.

As a good example of a ternary relationship, consider entity sets Parts, Suppliers, and Departments, and a relationship set Contracts (with descriptive attribute *qty*) that involves all of them. A contract specifies that a supplier will supply (some quantity of) a part to a department. This relationship cannot be adequately captured by a collection of binary relationships (without the use of aggregation). With binary relationships, we can denote that a supplier 'can supply' certain parts, that a department 'needs' some

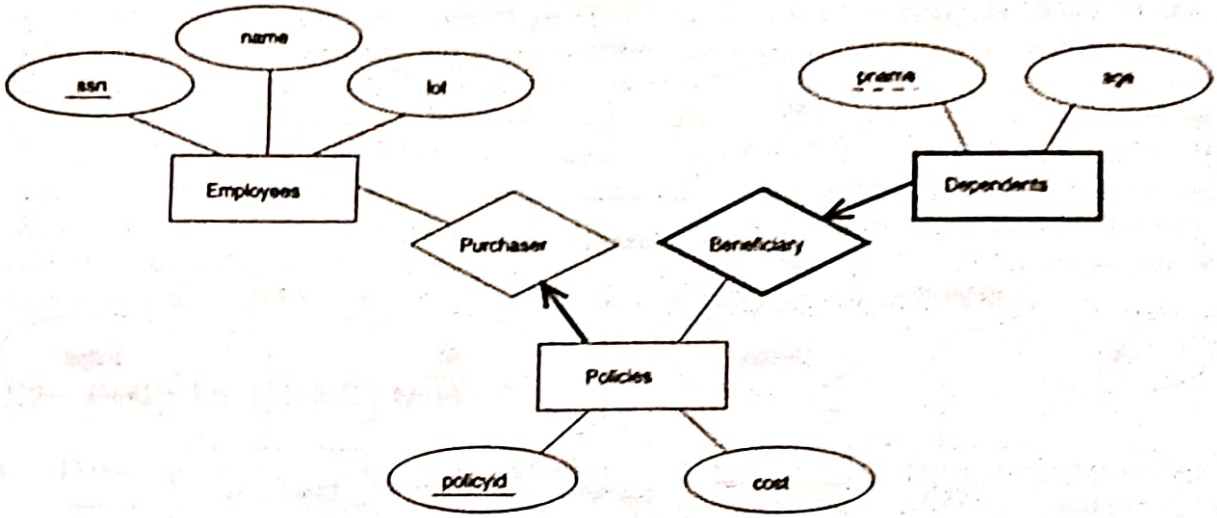

Figure 2.19 Policy Revisited

parts, or that a department 'deals with' a certain supplier. No combination of these relationships expresses the meaning of a contract adequately, for at least two reasons:

- The facts that supplier S can supply part P, that department D needs part P, and that D will buy from S do not necessarily imply that department D indeed buys part P from supplier S!
- We cannot represent the *qty* attribute of a contract cleanly.

2.5.4 Aggregation versus Ternary Relationships *

As we noted in Section 2.4.5, the choice between using aggregation or a ternary relationship is mainly determined by the existence of a relationship that relates a *relationship set* to an entity set (or second relationship set). The choice may also be guided by certain integrity constraints that we want to express. For example, consider the ER diagram shown in Figure 2.13. According to this diagram, a project can be sponsored by any number of departments, a department can sponsor one or more projects, and each sponsorship is monitored by one or more employees. If we don't need to record the *until* attribute of Monitors, then we might reasonably use a ternary relationship, say, Sponsors2, as shown in Figure 2.20.

Consider the constraint that each sponsorship (of a project by a department) be monitored by at most one employee. We cannot express this constraint in terms of the Sponsors2 relationship set. On the other hand, we can easily express the constraint by drawing an arrow from the aggregated relationship Sponsors to the relationship

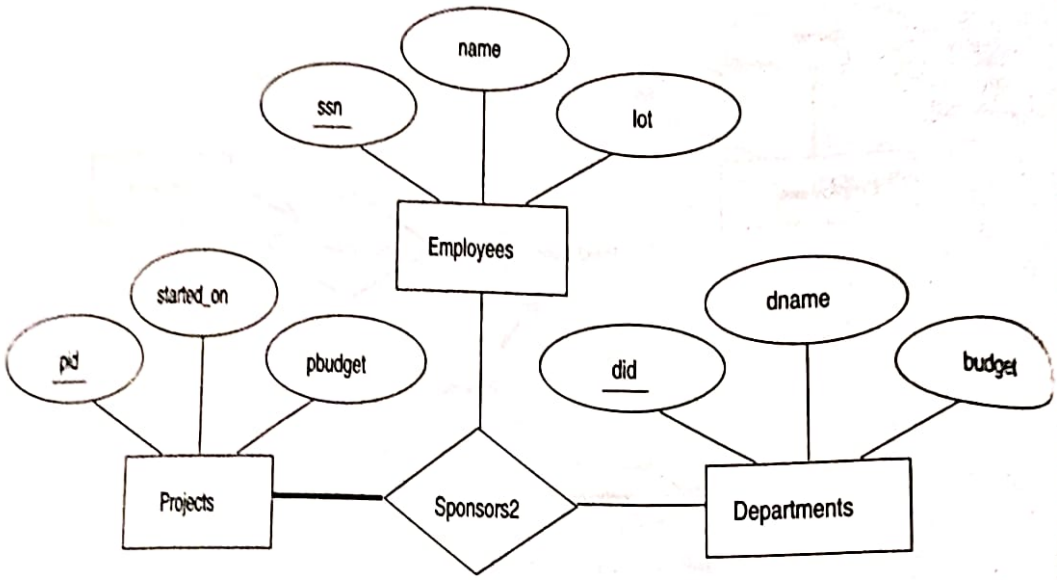

Figure 2.20 Using a Ternary Relationship instead of Aggregation

Monitors in Figure 2.13. Thus, the presence of such a constraint serves as another reason for using aggregation rather than a ternary relationship set.

2.6 CONCEPTUAL DESIGN FOR LARGE ENTERPRISES *

We have thus far concentrated on the constructs available in the ER model for describing various application concepts and relationships. The process of conceptual design consists of more than just describing small fragments of the application in terms of ER diagrams. For a large enterprise, the design may require the efforts of more than one designer and span data and application code used by a number of user groups. Using a high-level, semantic data model such as ER diagrams for conceptual design in such an environment offers the additional advantage that the high-level design can be diagrammatically represented and is easily understood by the many people who must provide input to the design process.

2.4.2

An important aspect of the design process is the methodology used to structure the development of the overall design and to ensure that the design takes into account all user requirements and is consistent. The usual approach is that the requirements of various user groups are considered, any conflicting requirements are somehow resolved, and a single set of global requirements is generated at the end of the requirements analysis phase. Generating a single set of global requirements is a difficult task, but it allows the conceptual design phase to proceed with the development of a logical schema that spans all the data and applications throughout the enterprise.

An alternative approach is to develop separate conceptual schemas for different user groups and to then *integrate* these conceptual schemas. To integrate multiple conceptual schemas, we must establish correspondences between entities, relationships, and attributes, and we must resolve numerous kinds of conflicts (e.g., naming conflicts, domain mismatches, differences in measurement units). This task is difficult in its own right. In some situations schema integration cannot be avoided—for example, when one organization merges with another, existing databases may have to be integrated. Schema integration is also increasing in importance as users demand access to *heterogeneous* data sources, often maintained by different organizations.

2.7 POINTS TO REVIEW

- Database design has six steps: requirements analysis, conceptual database design, logical database design, schema refinement, physical database design, and security design. Conceptual design should produce a high-level description of the data, and the entity-relationship (ER) data model provides a graphical approach to this design phase. **(Section 2.1)**
- In the ER model, a real-world object is represented as an *entity*. An *entity set* is a collection of structurally identical entities. Entities are described using *attributes*. Each entity set has a distinguished set of attributes called a *key* that can be used to uniquely identify each entity. **(Section 2.2)**
- A *relationship* is an association between two or more entities. A *relationship set* is a collection of relationships that relate entities from the same entity sets. A relationship can also have *descriptive attributes*. **(Section 2.3)**
- A *key constraint* between an entity set S and a relationship set restricts instances of the relationship set by requiring that each entity of S participate in at most one relationship. A *participation constraint* between an entity set S and a relationship set restricts instances of the relationship set by requiring that each entity of S participate in at least one relationship. The identity and existence of a *weak entity* depends on the identity and existence of another (*owner*) entity. *Class hierarchies* organize structurally similar entities through inheritance into sub- and superclasses. *Aggregation* conceptually transforms a relationship set into an entity set such that the resulting construct can be related to other entity sets. **(Section 2.4)**
- Development of an ER diagram involves important modeling decisions. A thorough understanding of the problem being modeled is necessary to decide whether to use an attribute or an entity set, an entity or a relationship set, a binary or ternary relationship, or aggregation. **(Section 2.5)**
- Conceptual design for large enterprises is especially challenging because data from many sources, managed by many groups, is involved. **(Section 2.6)**