# Data Abstraction and Data Independence

Database systems comprise of complex data-structures. In order to make the system efficient in terms of retrieval of data, and reduce complexity in terms of usability of users, developers use abstraction i.e. hide irrelevant details from the users. This approach simplifies database design.

There are mainly **3**levels of data abstraction:

|  |  |
| --- | --- |
| **Physical**: This is the lowest level of data abstraction. It tells us how the data is actually stored in memory. The access methods like sequential or random access and file organisation methods like B+ trees, hashing used for the same. Usability, size of memory, and the number of times the records are factors which we need to know while designing the database. Suppose we need to store the details of an employee. Blocks of storage and the amount of memory used for these purposes is kept hidden from the user. | https://contribute.geeksforgeeks.org/wp-content/uploads/13-1.png |

**Logical**: This level comprises of the information that is actually stored in the database in the form of tables. It also stores the relationship among the data entities in relatively simple structures. At this level, the information available to the user at the view level is unknown.  
We can store the various attributes of an employee and relationships, e.g. with the manager can also be stored.

**View**: This is the highest level of abstraction. Only a part of the actual database is viewed by the users. This level exists to ease the accessibility of the database by an individual user. Users view data in the form of rows and columns. Tables and relations are used to store data. Multiple views of the same database may exist. Users can just view the data and interact with the database, storage and implementation details are hidden from them.

The main purpose of data abstraction is achieving data independence in order to save time and cost required when the database is modified or altered.  
We have namely two levels of data independence arising from these levels of abstraction

**Physical level data independence** : It refers to the characteristic of being able to modify the physical schema without any alterations to the conceptual or logical schema, done for optimisation purposes, e.g., Conceptual structure of the database would not be affected by any change in storage size of the database system server. Changing from sequential to random access files is one such example.These alterations or modifications to the physical structure may include:

* Utilising new storage devices.
* Modifying data structures used for storage.
* Altering indexes or using alternative file organisation techniques etc.

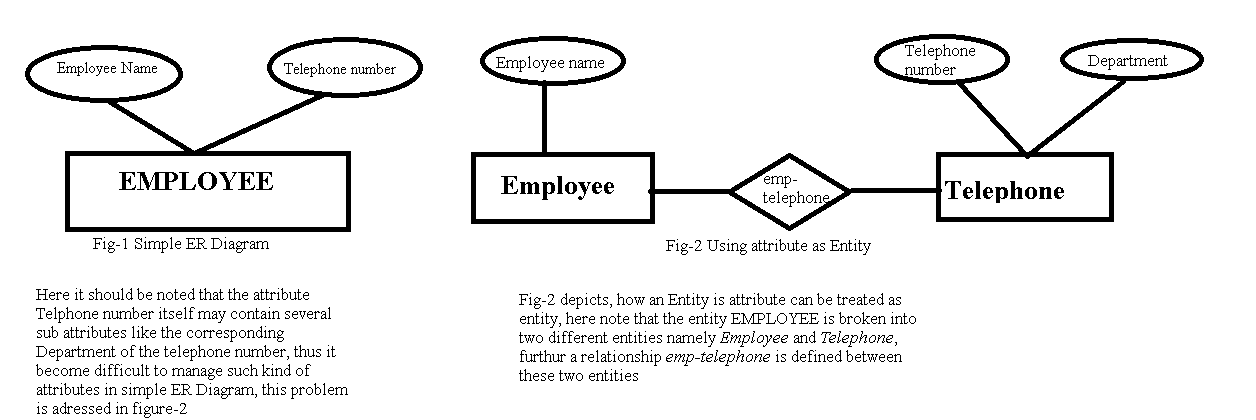
**Logical level data independence:** It refers characteristic of being able to modify the logical schema without affecting the external schema or application program. The user view of the data would not be affected by any changes to the conceptual view of the data. These changes may include insertion or deletion of attributes, altering table structures entities or relationships to the logical schema etc.

**Design Issues**

* Use of entity sets vs. attributes
  + Choice mainly depends on the structure of the enterprise being modeled, and on the semantics associated with the attribute in question.
* Use of entity sets vs. relationship sets
  + Possible guideline is to designate a relationship set to describe an action that occurs between entities
* Binary versus n-ary relationship sets
  + Although it is possible to replace any nonbinary (n-ary, for n > 2) relationship set by a number of distinct binary relationship sets, a nary relationship set shows more clearly that several entities participate in a single relationship.
* !Placement of relationship attributes

Design issue” we mean the basic issues that may arise while designing an E-R database schema and proper approach to solve them, so the basic issues and relative solutions concerning them are :

1. **Use of entity set vs attributes** - In these cases we treat an attribute as an entity for example - Consider the entity set *employee*with attributes *employee-name*and *telephone-number*.  
   It can easily be argued that a telephone is an entity in its own right with attributes *telephone-number*and *location*(the office where the telephone is located). If we take this point of view, we must redefine the *employee*entity set as:  
   *•*The *employee*entity set with attribute *employee-name*  
   *•*The *telephone*entity set with attributes *telephone-number*and *location*  
   *•*The relationship set *emp-telephone*, which denotes the association between employees and the telephones that they have. Such a conversion of attribute helps to give extra information about it when required.



The telephone number attribute itself may contain several sub-attributes like the corresponding department of the telephone no’s, thus it become difficult to manage such kind of attributes in ER Diagram, tis problem is addressed in fig2.

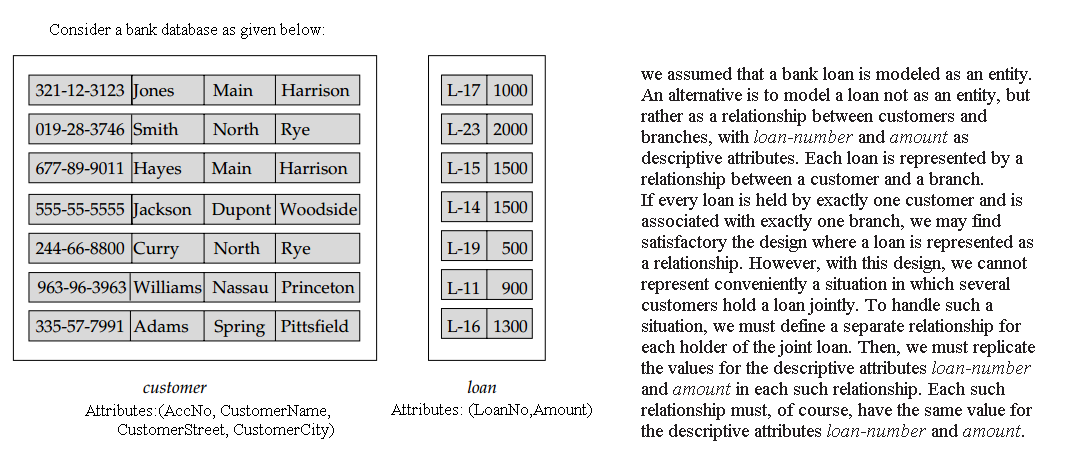
Fig2 depicts , how an entity attribute is treated as entity, here entity employee is broken into two entities employee ans telephone and relationship emp-tele is defined between two.

II way ; Consider the entity set employee with attributes employee-name and telephone-number. It can easily be argued that a telephone is an entity in its own right with attributes telephone-number and location (the office where the telephone is located). If we take this point of view, we must redefine the employee entity set as:

* The employee entity set with attribute employee-name
* The telephone entity set with attributes telephone-number and location
* The relationship set emp-telephone, which denotes the association between employees and the telephones that they have

It is the main difference between these two definitions of an   
employee? Treating a telephone as an attribute telephone-number implies that employees have precisely one telephone number each. Treating a telephone as an entity telephone permits employee to have several telephone numbers (including zero) associated with them. However, we could instead easily define telephone-number as a multivalued attribute to allow multiple telephones per employee.  
     The main difference then is that treating a telephone as an entity better models a situation where one may want to keep extra information about a telephone, such as its location, or its type (mobile, video phone or plain old telephone), or who all share the telephone. Thus treating telephone as an attribute and is appropriate when the generality may be useful.

1. **Use of Entity Sets vs Relationship Sets -**At several instances, It is not always clear whether an object can be best expressed by an entity set or a relationship set. This approach is used to break such dilemma where a set of relationship is defined between entities with designated action that has to occur between them. This approach can also be useful in deciding whether certain attributes may be more appropriately expressed as relationships.

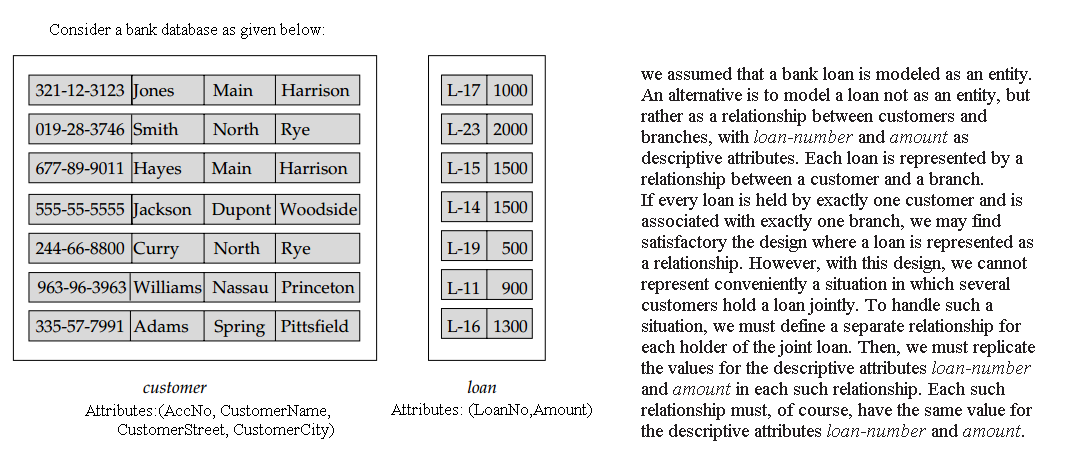


so, It is not always clear whether an object is best expressed by an entity set or a relationship. We assumed that a bank loan is modeled as an entity. An alternative is to model a loan not as an entity, but rather as a relationship between customer and branches, with loan number and amount as descriptive attributes. Each loan is represented by a relationship between a customer and a branch.  
       
     If every loan is held by exactly one customer and is associated with exactly one branch, we may find satisfactory the design where a loan is represented as a relationship. We must define a separate relationship for each holder of the joint loan. Then we must replicate the values for the descriptive attributes loan-number and amount in each such relationship. Each such relationship must, of course, have the same value for the descriptive attributes loan-number and amount.

**Two problems arise as a result of the replication:**

* the data are stored multiple times, wasting storage space and
* Updates potentially leave the data in an inconsistent state, where the values differ in two relationships for attributes that are supposed to have the same value.

 The problem of replication of the attributes loan-number and  
amount is absent in the original design, because there loan is an entity set.



1. **Binary versus nary Relationship Sets** - **Binary versus n-ary relationship sets**

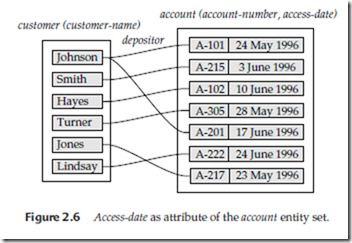
Relationships in databases are often binary. Some relationships that appear to be non binary could actually be better represented by several binary relationships. For instance one could create a ternary relationship parent, relating a child to his/her mother and father. However, such a relationship could also be represented by two binary relationships, mother and father, relating a child to his/her mother and father separately. Using the two relationships mother and father allows us record a child’s mother, even if we are not aware of the father’s identity; a null value would be required if the ternary relationship parent is used. Using binary relationship sets is preferable in this case.  
In fact, it is always possible to replace a non binary (n-ary, for n>2) relationship set by a number of distinct binary relationship sets.  
For simplicity, consider the abstract ternary (n=3) relationship set R, relating entity sets A, B and C. We replace the relationship set R by an entity set E, and create three relationship sets:

* RA, relating E and A
* RB, relating E and B
* RC, relating E and C

 If the relationship set R had any attributes, these are   
Assigned to entity set E; further, a special identifying attribute is created for E(since it must be possible to distinguish different entities in an entity set on the basis of their attribute values).

1. **Placement of Relationship Attributes -**

The cardinality ratio of a relationship can affect the placement of relationship at- tributes. Thus, attributes of one-to-one or one-to-many relationship sets can be associated with one of the participating entity sets, rather than with the relationship set. For instance, let us specify that *depositor*is a one-to-many relationship set such that one customer may have several accounts, but each account is held by only one customer. In this case, the attribute *access-date*, which speciﬁes when the customer last accessed that account, could be associated with the *account*entity set, as Figure 2.6 depicts; to keep the ﬁgure simple, only some of the attributes of the two entity sets are shown. Since each *account*entity participates in a relationship with at most one in- stance of *customer*, making this attribute designation would have the same meaning

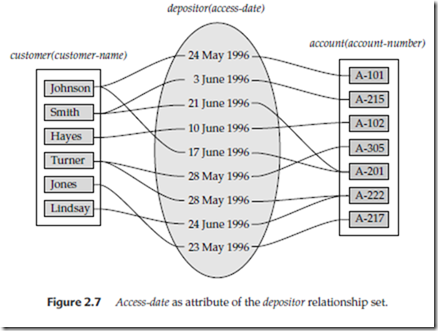
[](http://lh3.googleusercontent.com/-W3mI4pdUzGs/VT_D52hGXkI/AAAAAAABo1A/-z1A9CLXMnQ/s1600-h/image%5b2%5d.png)

as would placing *access-date*with the *depositor*relationship set. Attributes of a one-to- many relationship set can be repositioned to only the entity set on the “many” side of the relationship. For one-to-one relationship sets, on the other hand, the relationship attribute can be associated with either one of the participating entities.

The design decision of where to place descriptive attributes in such cases — as a relationship or entity attribute — should reﬂect the characteristics of the enterprise being modeled. The designer may choose to retain *access-date*as an attribute of *depositor*to express explicitly that an access occurs at the point of interaction between the *customer*and *account*entity sets.

The choice of attribute placement is more clear-cut for many-to-many relationship sets. Returning to our example, let us specify the perhaps more realistic case that *depositor*is a many-to-many relationship set expressing that a customer may have one or more accounts, and that an account can be held by one or more customers.

If we are to express the date on which a speciﬁc customer last accessed a speciﬁc account, *access-date*must be an attribute of the *depositor*relationship set, rather than either one of the participating entities. If *access-date*were an attribute of *account*, for instance, we could not determine which customer made the most recent access to a joint account. When an attribute is determined by the combination of participating entity sets, rather than by either entity separately, that attribute must be associated with the many-to-many relationship set. Figure 2.7 depicts the placement of *access-date*as a relationship attribute; again, to keep the ﬁgure simple, only some of the attributes of the two entity sets are shown.

[](http://lh3.googleusercontent.com/-Gq7bd0_waWU/VT_EYKeoiDI/AAAAAAABo1Q/PPm_pZPD81M/s1600-h/image%5b5%5d.png)

#### II copy D*esign Issues*

The notions of an entity set and a relationship set are not precise, and it is possible to deﬁne a set of entities and the relationships among them in a number of different ways. In this section, we examine basic issues in the design of an E-R database schema. Section 2.7.4 covers the design process in further detail.

***Use of Entity Sets versus Attributes***

Consider the entity set *employee*with attributes *employee-name*and *telephone-number*. It can easily be argued that a telephone is an entity in its own right with attributes *telephone-number*and *location*(the ofﬁce where the telephone is located). If we take this point of view, we must redeﬁne the *employee*entity set as:

• The *employee*entity set with attribute *employee-name*

• The *telephone*entity set with attributes *telephone-number*and *location*

• The relationship set *emp-telephone*, which denotes the association between employees and the telephones that they have

What, then, is the main difference between these two deﬁnitions of an employee? Treating a telephone as an attribute *telephone-number*implies that employees have precisely one telephone number each. Treating a telephone as an entity *telephone*per- mits employees to have several telephone numbers (including zero) associated with them. However, we could instead easily deﬁne *telephone-number*as a multivalued at- tribute to allow multiple telephones per employee.

The main difference then is that treating a telephone as an entity better models a situation where one may want to keep extra information about a telephone, such as its location, or its type (mobile, video phone, or plain old telephone), or who all share the telephone. Thus, treating telephone as an entity is more general than treating it as an attribute and is appropriate when the generality may be useful.

In contrast, it would not be appropriate to treat the attribute *employee-name*as an entity; it is difﬁcult to argue that *employee-name*is an entity in its own right (in contrast to the telephone). Thus, it is appropriate to have *employee-name*as an attribute of the *employee*entity set.

Two natural questions thus arise: What constitutes an attribute, and what constitutes an entity set? Unfortunately, there are no simple answers. The distinctions mainly depend on the structure of the real-world enterprise being modeled, and on the semantics associated with the attribute in question.

A common mistake is to use the primary key of an entity set as an attribute of another entity set, instead of using a relationship. For example, it is incorrect to model *customer-id*as an attribute of *loan*even if each loan had only one customer. The relationship *borrower*is the correct way to represent the connection between loans and customers, since it makes their connection explicit, rather than implicit via an attribute.

Another related mistake that people sometimes make is to designate the primary key attributes of the related entity sets as attributes of the relationship set. This should not be done, since the primary key attributes are already implicit in the relationship.

***Use of Entity Sets versus Relationship Sets***

It is not always clear whether an object is best expressed by an entity set or a relationship set. In Section 2.1.1, we assumed that a bank loan is modeled as an entity. An alternative is to model a loan not as an entity, but rather as a relationship between customers and branches, with *loan-number*and *amount*as descriptive attributes. Each loan is represented by a relationship between a customer and a branch.

If every loan is held by exactly one customer and is associated with exactly one branch, we may ﬁnd satisfactory the design where a loan is represented as a relationship. However, with this design, we cannot represent conveniently a situation in which several customers hold a loan jointly. To handle such a situation, we must de- ﬁne a separate relationship for each holder of the joint loan. Then, we must replicate the values for the descriptive attributes *loan-number*and *amount*in each such relation- ship. Each such relationship must, of course, have the same value for the descriptive attributes *loan-number*and *amount*.

Two problems arise as a result of the replication: (1) the data are stored multiple times, wasting storage space, and (2) updates potentially leave the data in an inconsistent state, where the values differ in two relationships for attributes that are supposed to have the same value. The issue of how to avoid such replication is treated formally by *normalization theory*, discussed in Chapter 7.

The problem of replication of the attributes *loan-number*and *amount*is absent in the original design of Section 2.1.1, because there *loan*is an entity set.

One possible guideline in determining whether to use an entity set or a relation- ship set is to designate a relationship set to describe an action that occurs between entities. This approach can also be useful in deciding whether certain attributes may be more appropriately expressed as relationships.

***Binary versus nary Relationship Sets***

Relationships in databases are often binary. Some relationships that appear to be nonbinary could actually be better represented by several binary relationships. For instance, one could create a ternary relationship *parent*, relating a child to his/her mother and father. However, such a relationship could also be represented by two binary relationships, *mother*and *father*, relating a child to his/her mother and father separately. Using the two relationships *mother*and *father*allows us record a child’s mother, even if we are not aware of the father’s identity; a null value would be required if the ternary relationship *parent*is used. Using binary relationship sets is preferable in this case.

In fact, it is always possible to replace a nonbinary (*n*-ary, for *n >*2) relationship set by a number of distinct binary relationship sets. For simplicity, consider the abstract ternary (*n*= 3) relationship set *R*, relating entity sets *A*, *B*, and *C*. We replace the relationship set *R*by an entity set *E*, and create three relationship sets:

• *RA*, relating *E*and *A*

• *RB*, relating *E*and *B*

• *RC*, relating *E*and *C*

If the relationship set *R*had any attributes, these are assigned to entity set *E*; further, a special identifying attribute is created for *E*(since it must be possible to distinguish different entities in an entity set on the basis of their attribute values). For each relationship (*ai, bi, ci*) in the relationship set *R*, we create a new entity *ei*in the entity set *E*. Then, in each of the three new relationship sets, we insert a relationship as follows:

• (*ei, ai*) in *RA*

• (*ei, bi*) in *RB*

• (*ei, ci*) in *RC*

We can generalize this process in a straightforward manner to *n*-ary relationship sets. Thus, conceptually, we can restrict the E-R model to include only binary relationship sets. However, this restriction is not always desirable.

• An identifying attribute may have to be created for the entity set created to represent the relationship set. This attribute, along with the extra relationship sets required, increases the complexity of the design and (as we shall see in Section 2.9) overall storage requirements.

• A *n*-ary relationship set shows more clearly that several entities participate in a single relationship.

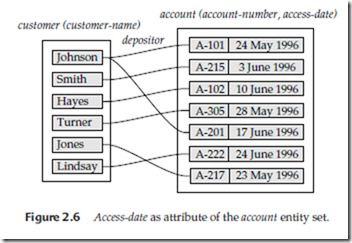
• There may not be a way to translate constraints on the ternary relationship into constraints on the binary relationships. For example, consider a constraint that says that *R*is many-to-one from *A, B*to *C*; that is, each pair of entities from *A*and *B*is associated with at most one *C*entity. This constraint cannot be expressed by using cardinality constraints on the relationship sets *RA, RB*, and *RC*.

Consider the relationship set *works-on*in Section 2.1.2, relating *employee*, *branch*, and *job*. We cannot directly split *works-on*into binary relationships between *employee*and *branch*and between *employee*and *job*. If we did so, we would be able to record that Jones is a manager and an auditor and that Jones works at Perryridge and Down- town; however, we would not be able to record that Jones is a manager at Perryridge and an auditor at Downtown, but is not an auditor at Perryridge or a manager at Downtown.

The relationship set *works-on*can be split into binary relationships by creating a new entity set as described above. However, doing so would not be very natural.

***Placement of Relationship Attributes***

The cardinality ratio of a relationship can affect the placement of relationship at- tributes. Thus, attributes of one-to-one or one-to-many relationship sets can be associated with one of the participating entity sets, rather than with the relationship set. For instance, let us specify that *depositor*is a one-to-many relationship set such that one customer may have several accounts, but each account is held by only one customer. In this case, the attribute *access-date*, which speciﬁes when the customer last accessed that account, could be associated with the *account*entity set, as Figure 2.6 depicts; to keep the ﬁgure simple, only some of the attributes of the two entity sets are shown. Since each *account*entity participates in a relationship with at most one in- stance of *customer*, making this attribute designation would have the same meaning

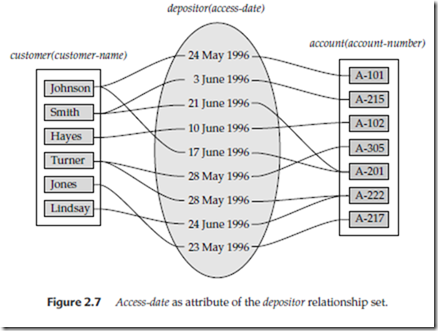
[](http://lh3.googleusercontent.com/-W3mI4pdUzGs/VT_D52hGXkI/AAAAAAABo1A/-z1A9CLXMnQ/s1600-h/image%5b2%5d.png)

as would placing *access-date*with the *depositor*relationship set. Attributes of a one-to- many relationship set can be repositioned to only the entity set on the “many” side of the relationship. For one-to-one relationship sets, on the other hand, the relationship attribute can be associated with either one of the participating entities.

The design decision of where to place descriptive attributes in such cases — as a relationship or entity attribute — should reﬂect the characteristics of the enterprise being modeled. The designer may choose to retain *access-date*as an attribute of *depositor*to express explicitly that an access occurs at the point of interaction between the *customer*and *account*entity sets.

The choice of attribute placement is more clear-cut for many-to-many relationship sets. Returning to our example, let us specify the perhaps more realistic case that *depositor*is a many-to-many relationship set expressing that a customer may have one or more accounts, and that an account can be held by one or more customers.

If we are to express the date on which a speciﬁc customer last accessed a speciﬁc account, *access-date*must be an attribute of the *depositor*relationship set, rather than either one of the participating entities. If *access-date*were an attribute of *account*, for instance, we could not determine which customer made the most recent access to a joint account. When an attribute is determined by the combination of participating entity sets, rather than by either entity separately, that attribute must be associated with the many-to-many relationship set. Figure 2.7 depicts the placement of *access-date*as a relationship attribute; again, to keep the ﬁgure simple, only some of the attributes of the two entity sets are shown.

[](http://lh3.googleusercontent.com/-Gq7bd0_waWU/VT_EYKeoiDI/AAAAAAABo1Q/PPm_pZPD81M/s1600-h/image%5b5%5d.png)

#### Extended E-R Features

Although the basic E-R concepts can model most database features, some aspects of a database may be more aptly expressed by certain extensions to the basic E-R model. In this section, we discuss the extended E-R features of specialization, generalization, higher- and lower-level entity sets, attribute inheritance, and aggregation.

##### Specialization

An entity set may include subgroupings of entities that are distinct in some way from other entities 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. The E-R model provides a means for representing these distinctive entity groupings.

Consider an entity set *person*, with attributes *name*, *street*, and *city*. A person may be further classiﬁed as one of the following:

• *customer*

• *employee*

Each of these person types is described by a set of attributes that includes all the at- tributes of entity set *person*plus possibly additional attributes. For example, *customer*entities may be described further by the attribute *customer-id*, whereas *employee*enti- ties may be described further by the attributes *employee-id*and *salary*. The process of designating subgroupings within an entity set is called **specialization**. The specialization of *person*allows us to distinguish among persons according to whether they are employees or customers.

As another example, suppose the bank wishes to divide accounts into two categories, checking account and savings account. Savings accounts need a minimum balance, but the bank may set interest rates differently for different customers, offering better rates to favored customers. Checking accounts have a ﬁxed interest rate, but offer an overdraft facility; the overdraft amount on a checking account must be recorded.

The bank could then create two specializations of *account*, namely *savings-account*and *checking-account*. As we saw earlier, account entities are described by the attributes *account-number*and *balance*. The entity set *savings-account*would have all the attributes of *account*and an additional attribute *interest-rate*. The entity set *checking account*would have all the attributes of *account*, and an additional attribute *overdraft amount*.

We can apply specialization repeatedly to reﬁne a design scheme. For instance, bank employees may be further classiﬁed as one of the following:

• *ofﬁcer*

• *teller*

• *secretary*

Each of these employee types is described by a set of attributes that includes all the attributes of entity set *employee*plus additional attributes. For example, *ofﬁcer*entities may be described further by the attribute *ofﬁce-number*, *teller*entities by the attributes *station-number*and *hours-per-week*, and *secretary*entities by the attribute *hours-per- week*. Further, *secretary*entities may participate in a relationship *secretary-for*, which identiﬁes which employees are assisted by a secretary.

An entity set may be specialized by more than one distinguishing feature. In our example, the distinguishing feature among employee entities is the job the employee performs. Another, coexistent, specialization could be based on whether the person is a temporary (limited-term) employee or a permanent employee, resulting in the entity sets *temporary-employee*and *permanent-employee*. When more than one specialization is formed on an entity set, a particular entity may belong to multiple specializations. For instance, a given employee may be a temporary employee who is a secretary.

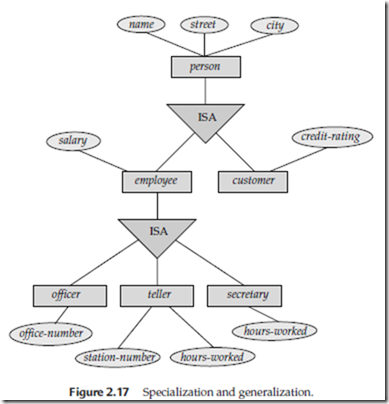
In terms of an E-R diagram, specialization is depicted by a *triangle*component labeled **ISA**, as Figure 2.17 shows. The label ISA stands for “is a” and represents, for example, that a customer “is a” person. The ISA relationship may also be referred to as a **superclass-subclass**relationship. Higher- and lower-level entity sets are depicted as regular entity sets — that is, as rectangles containing the name of the entity set.

##### Generalization

The reﬁnement from an initial entity set into successive levels of entity subgroupings represents a **top-down**design process in which distinctions are made explicit. The design process may also proceed in a **bottom-up**manner, in which multiple entity sets are synthesized into a higher-level entity set on the basis of common features. The database designer may have ﬁrst identiﬁed a *customer*entity set with the attributes *name*, *street*, *city*, and *customer-id*, and an *employee*entity set with the attributes *name*, *street*, *city*, *employee-id*, and *salary*.

There are similarities between the *customer*entity set and the *employee*entity set in the sense that they have several attributes in common. This commonality can be expressed by **generalization**, which is a containment relationship that exists between a *higher-level*entity set and one or more *lower-level*entity sets. In our example, *person*is the higher-level entity set and *customer*and *employee*are lower-level entity sets. Higher- and lower-level entity sets also may be designated by the terms **superclass**and **subclass**, respectively. The *person*entity set is the superclass of the *customer*and *employee*subclasses.

For all practical purposes, generalization is a simple inversion of specialization. We will apply both processes, in combination, in the course of designing the E-R

[](http://lh3.googleusercontent.com/-c2qTpCGAnCI/VT_Kk7StSOI/AAAAAAABo4I/17tsw8OHp3E/s1600-h/image%5b2%5d.png)

schema for an enterprise. In terms of the E-R diagram itself, we do not distinguish be- tween specialization and generalization. New levels of entity representation will be distinguished (specialization) or synthesized (generalization) as the design schema comes to express fully the database application and the user requirements of the database. Differences in the two approaches may be characterized by their starting point and overall goal.

Specialization stems from a single entity set; it emphasizes differences among entities within the set by creating distinct lower-level entity sets. These lower-level entity sets may have attributes, or may participate in relationships, that do not apply to all the entities in the higher-level entity set. Indeed, the reason a designer applies specialization is to represent such distinctive features. If *customer*and *employee*neither have attributes that *person*entities do not have nor participate in different relationships than those in which *person*entities participate, there would be no need to specialize the *person*entity set.

Generalization proceeds from the recognition that a number of entity sets share some common features (namely, they are described by the same attributes and participate in the same relationship sets). On the basis of their commonalities, generalization synthesizes these entity sets into a single, higher-level entity set. Generalization is used to emphasize the similarities among lower-level entity sets and to hide the differences; it also permits an economy of representation in that shared attributes are not repeated.

##### Attribute Inheritance

A crucial property of the higher- and lower-level entities created by specialization and generalization is **attribute inheritance**. The attributes of the higher-level entity sets are said to be **inherited**by the lower-level entity sets. For example, *customer*and *employee*inherit the attributes of *person*. Thus, *customer*is described by its *name*, *street*, and *city*attributes, and additionally a *customer-id*attribute; *employee*is described by its *name*, *street*, and *city*attributes, and additionally *employee-id*and *salary*attributes.

A lower-level entity set (or subclass) also inherits participation in the relationship sets in which its higher-level entity (or superclass) participates. The *ofﬁcer*, *teller*, and *secretary*entity sets can participate in the *works-for*relationship set, since the superclass *employee*participates in the *works-for*relationship. Attribute inheritance applies through all tiers of lower-level entity sets. The above entity sets can participate in any relationships in which the *person*entity set participates.

Whether a given portion of an E-R model was arrived at by specialization or generalization, the outcome is basically the same:

• A higher-level entity set with attributes and relationships that apply to all of its lower-level entity sets

• Lower-level entity sets with distinctive features that apply only within a particular lower-level entity set

In what follows, although we often refer to only generalization, the properties that we discuss belong fully to both processes.

Figure 2.17 depicts a **hierarchy**of entity sets. In the ﬁgure, *employee*is a lower-level entity set of *person*and a higher-level entity set of the *ofﬁcer, teller*, and *secretary*entity sets. In a hierarchy, a given entity set may be involved as a lower-level entity set in only one ISA relationship; that is, entity sets in this diagram have only **single inheritance**. If an entity set is a lower-level entity set in more than one ISA relationship, then the entity set has **multiple inheritance**, and the resulting structure is said to be a *lattice*.

##### Constraints on Generalizations

To model an enterprise more accurately, the database designer may choose to place certain constraints on a particular generalization. One type of constraint involves determining which entities can be members of a given lower-level entity set. Such membership may be one of the following:

• **Condition-deﬁned**. In condition-deﬁned lower-level entity sets, membership is evaluated on the basis of whether or not an entity satisﬁes an explicit condition or predicate. For example, assume that the higher-level entity set *account*has the attribute *account-type*. All *account*entities are evaluated on the deﬁning *account-type*attribute. Only those entities that satisfy the condition *account-type*= “savings account” are allowed to belong to the lower-level en- tity set *person*. All entities that satisfy the condition *account-type*= “checking account” are included in *checking account*. Since all the lower-level entities are evaluated on the basis of the same attribute (in this case, on *account-type*), this type of generalization is said to be **attribute-deﬁned**.

• **User-deﬁned**. User-deﬁned lower-level entity sets are not constrained by a membership condition; rather, the database user assigns entities to a given entity set. For instance, let us assume that, after 3 months of employment, bank employees are assigned to one of four work teams. We therefore represent the teams as four lower-level entity sets of the higher-level *employee*entity set. A given employee is not assigned to a speciﬁc team entity automatically on the basis of an explicit deﬁning condition. Instead, the user in charge of this decision makes the team assignment on an individual basis. The assignment is implemented by an operation that adds an entity to an entity set.

A second type of constraint relates to whether or not entities may belong to more than one lower-level entity set within a single generalization. The lower-level entity sets may be one of the following:

• **Disjoint**. A *disjointness constraint*requires that an entity belong to no more than one lower-level entity set. In our example, an *account*entity can satisfy only one condition for the *account-type*attribute; an entity can be either a savings account or a checking account, but cannot be both.

• **Overlapping**. In *overlapping generalizations*, the same entity may belong to more than one lower-level entity set within a single generalization. For an illustration, consider the employee work team example, and assume that certain managers participate in more than one work team. A given employee may therefore appear in more than one of the team entity sets that are lower-level entity sets of *employee*. Thus, the generalization is overlapping.

As another example, suppose generalization applied to entity sets *customer*and *employee*leads to a higher-level entity set *person*. The generalization is overlapping if an employee can also be a customer.

Lower-level entity overlap is the default case; a disjointness constraint must be placed explicitly on a generalization (or specialization). We can note a disjointedness constraint in an E-R diagram by adding the word *disjoint*next to the triangle symbol.

A ﬁnal constraint, the **completeness constraint**on a generalization or specializa- tion, speciﬁes whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within the generalization/specialization. This constraint may be one of the following:

• **Total generalization**or **specialization**. Each higher-level entity must belong to a lower-level entity set.

• **Partial generalization**or **specialization**. Some higher-level entities may not belong to any lower-level entity set.

Partial generalization is the default. We can specify total generalization in an E-R diagram by using a double line to connect the box representing the higher-level entity set to the triangle symbol. (This notation is similar to the notation for total participation in a relationship.)

The *account*generalization is total: All account entities must be either a savings account or a checking account. Because the higher-level entity set arrived at through generalization is generally composed of only those entities in the lower-level entity sets, the completeness constraint for a generalized higher-level entity set is usually total. When the generalization is partial, a higher-level entity is not constrained to appear in a lower-level entity set. The work team entity sets illustrate a partial specialization. Since employees are assigned to a team only after 3 months on the job, some *employee*entities may not be members of any of the lower-level team entity sets.

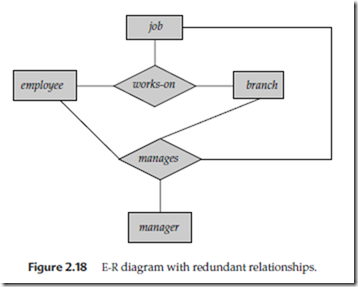
We may characterize the team entity sets more fully as a partial, overlapping specialization of *employee*. The generalization of *checking-account*and *savings-account*into *account*is a total, disjoint generalization. The completeness and disjointness constraints, however, do not depend on each other. Constraint patterns may also be partial-disjoint and total-overlapping.

We can see that certain insertion and deletion requirements follow from the constraints that apply to a given generalization or specialization. For instance, when a total completeness constraint is in place, an entity inserted into a higher-level entity set must also be inserted into at least one of the lower-level entity sets. With a condition-deﬁned constraint, all higher-level entities that satisfy the condition must be inserted into that lower-level entity set. Finally, an entity that is deleted from a higher-level entity set also is deleted from all the associated lower-level entity sets to which it belongs.

##### Aggregation

One limitation of the E-R model is that it cannot express relationships among relationships. To illustrate the need for such a construct, consider the ternary relationship *works-on*, which we saw earlier, between a *employee*, *branch*, and *job*(see Figure 2.13). Now, suppose we want to record managers for tasks performed by an employee at a branch; that is, we want to record managers for (*employee*, *branch*, *job*) combinations. Let us assume that there is an entity set *manager*.

One alternative for representing this relationship is to create a quaternary relation- ship *manages*between *employee*, *branch*, *job*, and *manager*. (A quaternary relationship is required — a binary relationship between *manager*and *employee*would not permit us to represent which (*branch*, *job*) combinations of an employee are managed by which manager.) Using the basic E-R modeling constructs, we obtain the E-R diagram of Figure 2.18. (We have omitted the attributes of the entity sets, for simplicity.)

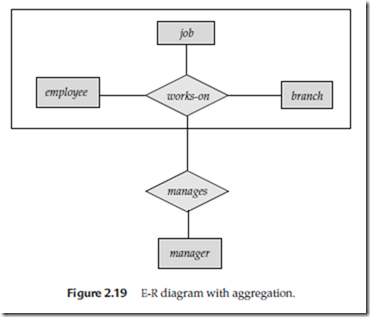
It appears that the relationship sets *works-on*and *manages*can be combined into one single relationship set. Nevertheless, we should not combine them into a single relationship, since some *employee*, *branch*, *job*combinations many not have a manager.[](http://lh3.googleusercontent.com/-CfhOKSMkJdg/VT_KvAGM4WI/AAAAAAABo4Y/2o8mNEIAO6I/s1600-h/image%5b5%5d.png)

There is redundant information in the resultant ﬁgure, however, since every *employee*, *branch*, *job*combination in *manages*is also in *works-on*. If the manager were a value rather than an *manager*entity, we could instead make *manager*a multivalued at- tribute of the relationship *works-on*. But doing so makes it more difﬁcult (logically as well as in execution cost) to ﬁnd, for example, employee-branch-job triples for which a manager is responsible. Since the manager is a *manager*entity, this alternative is ruled out in any case.

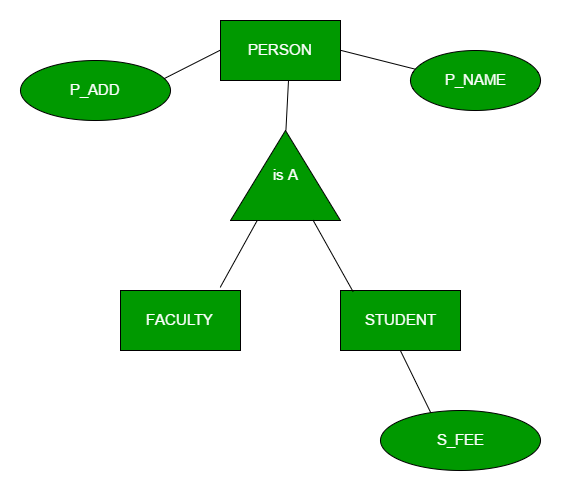
The best way to model a situation such as the one just described is to use aggregation. **Aggregation**is an abstraction through which relationships are treated as higher- level entities. Thus, for our example, we regard the relationship set *works-on*(relating the entity sets *employee*, *branch*, and *job*) as a higher-level entity set called *works-on*. Such an entity set is treated in the same manner as is any other entity set. We can then create a binary relationship *manages*between *works-on*and *manager*to represent who manages what tasks. Figure 2.19 shows a notation for aggregation commonly used to represent the above situation.

##### Alternative E-R Notations

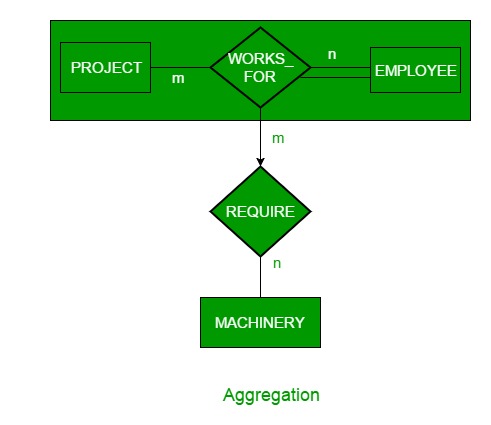
Figure 2.20 summarizes the set of symbols we have used in E-R diagrams. There is no universal standard for E-R diagram notation, and different books and E-R diagram software use different notations; Figure 2.21 indicates some of the alternative notations that are widely used. An entity set may be represented as a box with the name outside, and the attributes listed one below the other within the box. The primary key attributes are indicated by listing them at the top, with a line separating them from the other attributes.

[](http://lh3.googleusercontent.com/-8ya6xUZDppg/VT_K22QzgII/AAAAAAABo4o/4u47ZsTOTLM/s1600-h/image%5b8%5d.png)

Cardinality constraints can be indicated in several different ways, as Figure 2.21 shows. The labels *∗*and 1 on the edges out of the relationship are sometimes used for depicting many-to-many, one-to-one, and many-to-one relationships, as the ﬁgure shows. The case of one-to-many is symmetric to many-to-one, and is not shown. In another alternative notation in the ﬁgure, relationship sets are represented by lines between entity sets, without diamonds; only binary relationships can be modeled thus. Cardinality constraints in such a notation are shown by “crow’s foot” notation, as in the ﬁgure.

Ii copy **Generalization –**  
Generalization is the process of extracting common properties from a set of entities and create a generalized entity from it. It is a bottom-up approach in which two or more entities can be generalized to a higher level entity if they have some attributes in common. For Example, STUDENT and FACULTY can be generalized to a higher level entity called PERSON as shown in Figure 1. In this case, common attributes like P\_NAME, P\_ADD become part of higher entity (PERSON) and specialized attributes like S\_FEE become part of specialized entity (STUDENT).  
[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/generalization.png)

|  |  |
| --- | --- |
| **Specialization –** In specialization, an entity is divided into sub-entities based on their characteristics. It is a top-down approach where higher level entity is specialized into two or more lower level entities. For Example, EMPLOYEE entity in an Employee management system can be specialized into DEVELOPER, TESTER etc. as shown in Figure 2. In this case, common attributes like E\_NAME, E\_SAL etc. become part of higher entity (EMPLOYEE) and specialized attributes like TES\_TYPE become part of specialized entity | [img2](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/specialization.png) |

**Aggregation –**  
An ER diagram is not capable of representing relationship between an entity and a relationship which may be required in some scenarios. In those cases, a relationship with its corresponding entities is aggregated into a higher level entity. For Example, Employee working for a project may require some machinery. So, REQUIRE relationship is needed between relationship WORKS\_FOR and entity MACHINERY. Using aggregation, WORKS\_FOR relationship with its entities EMPLOYEE and PROJECT is aggregated into single entity and relationship REQUIRE is created between aggregated entity and MACHINERY.  
[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/aggregation.png)