P A R T 1

Data Models

A **data model** is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. In this part, we study two data models—the entity−relationship model and the relational model.

The entity−relationship (E-R) model is a high-level data model. It is based on a perception of a real world that consists of a collection of basic objects, called *entities*, and of *relationships* among these objects.

The relational model is a lower-level model. It uses a collection of tables to represent both data and the relationships among those data. Its conceptual simplicity has led to its widespread adoption; today a vast majority of database products are based on the relational model. Designers often formulate database schema design by first modeling data at a high level, using the E-R model, and then translating it into the the relational model.

We shall study other data models later in the book. The object-oriented data model, for example, extends the representation of entities by adding notions of encapsulation, methods (functions), and object identity. The object-relational data model combines features of the object-oriented data model and the relational data model. Chapters 8 and 9, respectively, cover these two data models.

C H A P T E R 2

Entity-Relationship Model

The **entity-relationship** (**E-R**) data model perceives the real world as consisting of basic objects, called *entities*, and *relationships* among these objects. It was developed to facilitate database design by allowing specification of an *enterprise schema*, which represents the overall logical structure of a database. The E-R data model is one of several semantic data models; the semantic aspect of the model lies in its representation of the meaning of the data. The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. Because of this usefulness, many database-design tools draw on concepts from the E-R model.

2.1 Basic Concepts

The E-R data model employs three basic notions: entity sets, relationship sets, and attributes.

2.1.1 Entity Sets

An **entity** is a "thing" or "object" in the real world that is distinguishable from all other objects. For example, each person in an enterprise is an entity. An entity has a set of properties, and the values for some set of properties may uniquely identify an entity. For instance, a person may have a *person-id* property whose value uniquely identifies that person. Thus, the value 677-89-9011 for *person-id* would uniquely identify one particular person in the enterprise. Similarly, loans can be thought of as entities, and loan number L-15 at the Perryridge branch uniquely identifies a loan entity. An entity may be concrete, such as a person or a book, or it may be abstract, such as a loan, or a holiday, or a concept.

An **entity set** is a set of entities of the same type that share the same properties, or attributes. The set of all persons who are customers at a given bank, for example, can be defined as the entity set *customer*. Similarly, the entity set *loan* might represent the set of all loans awarded by a particular bank. The individual entities that constitute a set are said to be the *extension* of the entity set. Thus, all the individual bank customers are the extension of the entity set *customer*.

Entity sets do not need to be disjoint. For example, it is possible to define the entity set of all employees of a bank (*employee*) and the entity set of all customers of the bank (*customer*). A *person* entity may be an *employee* entity, a *customer* entity, both, or neither.

An entity is represented by a set of **attributes**. Attributes are descriptive properties possessed by each member of an entity set. The designation of an attribute for an entity set expresses that the database stores similar information concerning each entity in the entity set; however, each entity may have its own value for each attribute. Possible attributes of the *customer* entity set are *customer-id*, *customer-name*, *customer-street*, and *customer-city*. In real life, there would be further attributes, such as street number, apartment number, state, postal code, and country, but we omit them to keep our examples simple. Possible attributes of the *loan* entity set are *loan-number* and *amount*.

Each entity has a **value** for each of its attributes. For instance, a particular *customer* entity may have the value 321-12-3123 for *customer-id*, the value Jones for *customer-name*, the value Main for *customer-street*, and the value Harrison for *customer-city*.

The *customer-id* attribute is used to uniquely identify customers, since there may be more than one customer with the same name, street, and city. In the United States, many enterprises find it convenient to use the *social-security* number of a person[1] as an attribute whose value uniquely identifies the person. In general the enterprise would have to create and assign a unique identifier for each customer.

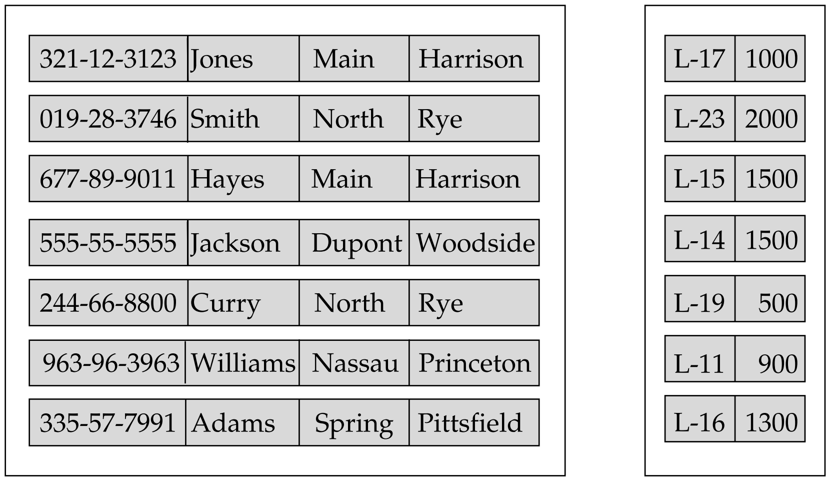
For each attribute, there is a set of permitted values, called the **domain**, or **value set**, of that attribute. The domain of attribute *customer-name* might be the set of all text strings of a certain length. Similarly, the domain of attribute *loan-number* might be the set of all strings of the form "L-n" where n is a positive integer.

A database thus includes a collection of entity sets, each of which contains any number of entities of the same type. Figure 2.1 shows part of a bank database that consists of two entity sets: *customer* and *loan*.

Formally, an attribute of an entity set is a function that maps from the entity set into a domain. Since an entity set may have several attributes, each entity can be described by a set of (attribute, data value) pairs, one pair for each attribute of the entity set. For example, a particular *customer* entity may be described by the set {(*customer-id*, 677-89-9011), (*customer-name*, Hayes), (*customer-street*, Main), (*customer-city*, Harrison)}, meaning that the entity describes a person named Hayes whose customer identifier is 677-89-9011 and who resides at Main Street in Harrison. We can see, at this point, an integration of the abstract schema with the actual enterprise being modeled. The attribute values describing an entity will constitute a significant portion of the data stored in the database.

An attribute, as used in the E-R model, can be characterized by the following attribute types.

1. In the United States, the government assigns to each person in the country a unique number, called a social-security number, to identify that person uniquely. Each person is supposed to have only one social-security number, and no two people are supposed to have the same social-security number.



321-12-3123	Jones	Main	Harrison
019-28-3746	Smith	North	Rye
677-89-9011	Hayes	Main	Harrison
555-55-5555	Jackson	Dupont	Woodside
244-66-8800	Curry	North	Rye
963-96-3963	Williams	Nassau	Princeton
335-57-7991	Adams	Spring	Pittsfield

customer

L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-19	500
L-11	900
L-16	1300

loan

Figure 2.1 Entity sets *customer* and *loan*.

- **Simple** and **composite** attributes. In our examples thus far, the attributes have been simple; that is, they are not divided into subparts. **Composite** attributes, on the other hand, can be divided into subparts (that is, other attributes). For example, an attribute *name* could be structured as a composite attribute consisting of *first-name*, *middle-initial*, and *last-name*. Using composite attributes in a design schema is a good choice if a user will wish to refer to an entire attribute on some occasions, and to only a component of the attribute on other occasions. Suppose we were to substitute for the *customer* entity-set attributes *customer-street* and *customer-city* the composite attribute *address* with the attributes *street*, *city*, *state*, and *zip-code*.[2] Composite attributes help us to group together related attributes, making the modeling cleaner.

  Note also that a composite attribute may appear as a hierarchy. In the composite attribute *address*, its component attribute *street* can be further divided into *street-number*, *street-name*, and *apartment-number*. Figure 2.2 depicts these examples of composite attributes for the *customer* entity set.

- **Single-valued** and **multivalued** attributes. The attributes in our examples all have a single value for a particular entity. For instance, the *loan-number* attribute for a specific loan entity refers to only one loan number. Such attributes are said to be **single valued**. There may be instances where an attribute has a set of values for a specific entity. Consider an *employee* entity set with the attribute *phone-number*. An employee may have zero, one, or several phone numbers, and different employees may have different numbers of phones. This type of attribute is said to be **multivalued**. As another example, an at-

2. We assume the address format used in the United States, which includes a numeric postal code called a zip code.

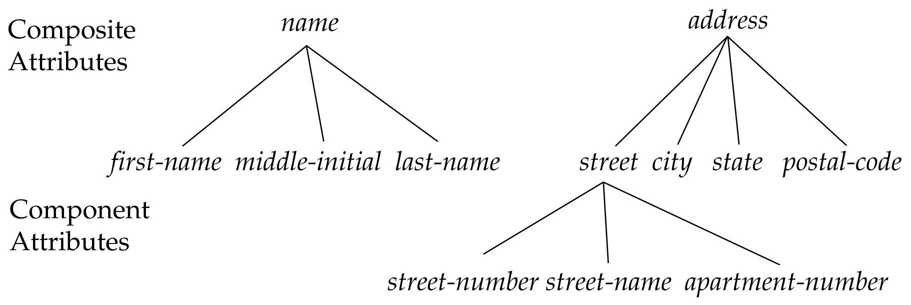


Figure 2.2 Composite attributes *customer-name* and *customer-address*.

tribute *dependent-name* of the *employee* entity set would be multivalued, since any particular employee may have zero, one, or more dependent(s).

Where appropriate, upper and lower bounds may be placed on the number of values in a multivalued attribute. For example, a bank may limit the number of phone numbers recorded for a single customer to two. Placing bounds in this case expresses that the *phone-number* attribute of the *customer* entity set may have between zero and two values.

- **Derived** attribute. The value for this type of attribute can be derived from the values of other related attributes or entities. For instance, let us say that the *customer* entity set has an attribute *loans-held*, which represents how many loans a customer has from the bank. We can derive the value for this attribute by counting the number of *loan* entities associated with that customer.

  As another example, suppose that the *customer* entity set has an attribute *age*, which indicates the customer's age. If the *customer* entity set also has an attribute *date-of-birth*, we can calculate *age* from *date-of-birth* and the current date. Thus, *age* is a derived attribute. In this case, *date-of-birth* may be referred to as a *base* attribute, or a *stored* attribute. The value of a derived attribute is not stored, but is computed when required.

An attribute takes a **null** value when an entity does not have a value for it. The *null* value may indicate "not applicable"—that is, that the value does not exist for the entity. For example, one may have no middle name. *Null* can also designate that an attribute value is unknown. An unknown value may be either *missing* (the value does exist, but we do not have that information) or *not known* (we do not know whether or not the value actually exists).

For instance, if the *name* value for a particular customer is *null*, we assume that the value is missing, since every customer must have a name. A null value for the *apartment-number* attribute could mean that the address does not include an apartment number (not applicable), that an apartment number exists but we do not know what it is (missing), or that we do not know whether or not an apartment number is part of the customer's address (unknown).

A database for a banking enterprise may include a number of different entity sets. For example, in addition to keeping track of customers and loans, the bank also provides accounts, which are represented by the entity set *account* with attributes *account-number* and *balance*. Also, if the bank has a number of different branches, then we may keep information about all the branches of the bank. Each *branch* entity set may be described by the attributes *branch-name*, *branch-city*, and *assets*.

2.1.2 Relationship Sets

A **relationship** is an association among several entities. For example, we can define a relationship that associates customer Hayes with loan L-15. This relationship specifies that Hayes is a customer with loan number L-15.

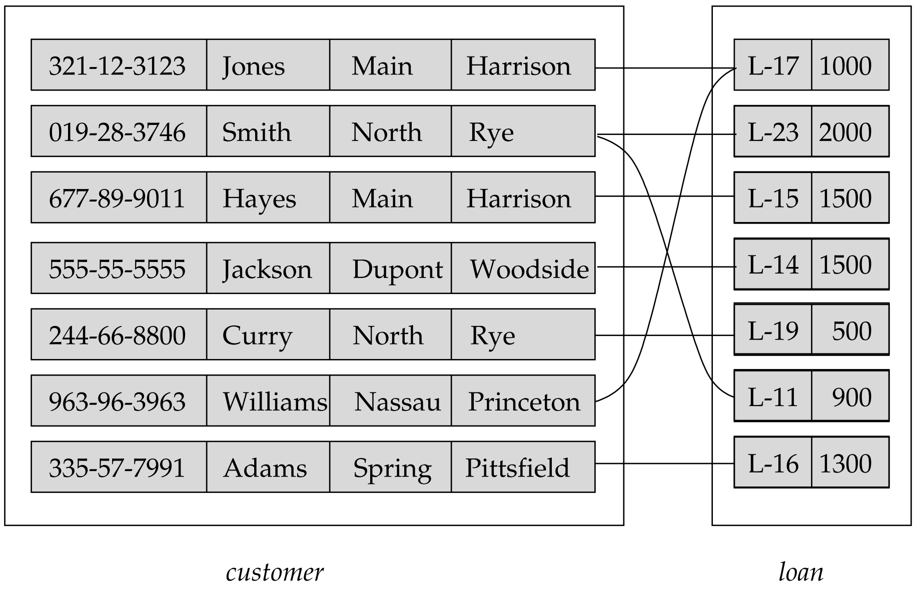
A **relationship set** is a set of relationships of the same type. Formally, it is a mathematical relation on $n \geq 2$ (possibly nondistinct) entity sets. If $E_1,\ E_2, \ldots, E_n$ are entity sets, then a relationship set *R* is a subset of

$$\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

where $(e_1, e_2, \ldots, e_n)$ is a relationship.

Consider the two entity sets *customer* and *loan* in Figure 2.1. We define the relationship set *borrower* to denote the association between customers and the bank loans that the customers have. Figure 2.3 depicts this association.

As another example, consider the two entity sets *loan* and *branch*. We can define the relationship set *loan-branch* to denote the association between a bank loan and the branch in which that loan is maintained.



321-12-3123 Jones Main Harrison
019-28-3746 Smith North Rye
677-89-9011 Hayes Main Harrison
555-55-5555 Jackson Dupont Woodside
244-66-8800 Curry North Rye
963-96-3963 Williams Nassau Princeton
335-57-7991 Adams Spring Pittsfield
L-17 1000
L-23 2000
L-15 1500
L-14 1500
L-19 500
L-11 900
L-16 1300
customer
loan

Figure 2.3 Relationship set *borrower*.

The association between entity sets is referred to as participation; that is, the entity sets $E_1,\ E_2, \ldots, E_n$ **participate** in relationship set R. A **relationship instance** in an E-R schema represents an association between the named entities in the real-world enterprise that is being modeled. As an illustration, the individual *customer* entity Hayes, who has customer identifier 677-89-9011, and the *loan* entity L-15 participate in a relationship instance of *borrower*. This relationship instance represents that, in the real-world enterprise, the person called Hayes who holds *customer-id* 677-89-9011 has taken the loan that is numbered L-15.

The function that an entity plays in a relationship is called that entity's **role**. Since entity sets participating in a relationship set are generally distinct, roles are implicit and are not usually specified. However, they are useful when the meaning of a relationship needs clarification. Such is the case when the entity sets of a relationship set are not distinct; that is, the same entity set participates in a relationship set more than once, in different roles. In this type of relationship set, sometimes called a **recursive** relationship set, explicit role names are necessary to specify how an entity participates in a relationship instance. For example, consider an entity set *employee* that records information about all the employees of the bank. We may have a relationship set *works-for* that is modeled by ordered pairs of *employee* entities. The first employee of a pair takes the role of *worker*, whereas the second takes the role of *manager*. In this way, all relationships of *works-for* are characterized by (worker, manager) pairs; (manager, worker) pairs are excluded.

A relationship may also have attributes called **descriptive attributes**. Consider a relationship set *depositor* with entity sets *customer* and *account*. We could associate the attribute *access-date* to that relationship to specify the most recent date on which a customer accessed an account. The *depositor* relationship among the entities corresponding to customer Jones and account A-217 has the value "23 May 2001" for attribute *access-date*, which means that the most recent date that Jones accessed account A-217 was 23 May 2001.

As another example of descriptive attributes for relationships, suppose we have entity sets *student* and *course* which participate in a relationship set *registered-for*. We may wish to store a descriptive attribute *for-credit* with the relationship, to record whether a student has taken the course for credit, or is auditing (or sitting in on) the course.

A relationship instance in a given relationship set must be uniquely identifiable from its participating entities, without using the descriptive attributes. To understand this point, suppose we want to model all the dates when a customer accessed an account. The single-valued attribute *access-date* can store a single access date only . We cannot represent multiple access dates by multiple relationship instances between the same customer and account, since the relationship instances would not be uniquely identifiable using only the participating entities. The right way to handle this case is to create a multivalued attribute *access-dates*, which can store all the access dates.

However, there can be more than one relationship set involving the same entity sets. In our example the *customer* and *loan* entity sets participate in the relationship set *borrower*. Additionally, suppose each loan must have another customer who serves as a guarantor for the loan. Then the *customer* and *loan* entity sets may participate in another relationship set, *guarantor*.

The relationship sets *borrower* and *loan-branch* provide an example of a **binary** relationship set—that is, one that involves two entity sets. Most of the relationship sets in a database system are binary. Occasionally, however, relationship sets involve more than two entity sets.

As an example, consider the entity sets *employee*, *branch*, and *job*. Examples of *job* entities could include manager, teller, auditor, and so on. Job entities may have the attributes *title* and *level*. The relationship set *works-on* among *employee*, *branch*, and *job* is an example of a ternary relationship. A ternary relationship among Jones, Perryridge, and manager indicates that Jones acts as a manager at the Perryridge branch. Jones could also act as auditor at the Downtown branch, which would be represented by another relationship. Yet another relationship could be between Smith, Downtown, and teller, indicating Smith acts as a teller at the Downtown branch.

The number of entity sets that participate in a relationship set is also the **degree** of the relationship set. A binary relationship set is of degree 2; a ternary relationship set is of degree 3.

2.2 Constraints

An E-R enterprise schema may define certain constraints to which the contents of a database must conform. In this section, we examine mapping cardinalities and participation constraints, which are two of the most important types of constraints.

2.2.1 Mapping Cardinalities

Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set.

Mapping cardinalities are most useful in describing binary relationship sets, although they can contribute to the description of relationship sets that involve more than two entity sets. In this section, we shall concentrate on only binary relationship sets.

For a binary relationship set *R* between entity sets *A* and *B*, the mapping cardinality must be one of the following:

- **One to one**. An entity in *A* is associated with *at most* one entity in *B*, and an entity in *B* is associated with *at most* one entity in *A*. (See Figure 2.4a.)
- **One to many**. An entity in *A* is associated with any number (zero or more) of entities in *B*. An entity in *B*, however, can be associated with *at most* one entity in *A*. (See Figure 2.4b.)
- **Many to one**. An entity in *A* is associated with *at most* one entity in *B*. An entity in *B*, however, can be associated with any number (zero or more) of entities in *A*. (See Figure 2.5a.)
- **Many to many**. An entity in *A* is associated with any number (zero or more) of entities in *B*, and an entity in *B* is associated with any number (zero or more) of entities in *A*. (See Figure 2.5b.)

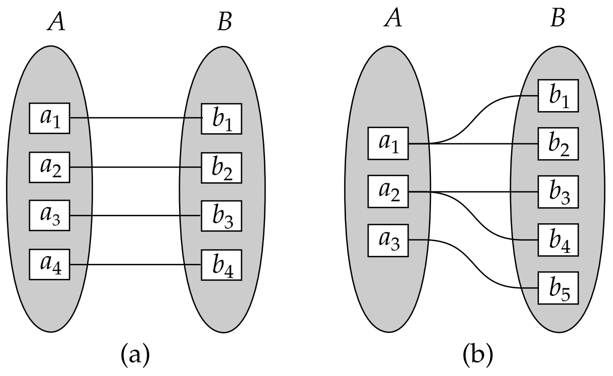


Figure 2.4 Mapping cardinalities. (a) One to one. (b) One to many.

The appropriate mapping cardinality for a particular relationship set obviously depends on the real-world situation that the relationship set is modeling.

As an illustration, consider the *borrower* relationship set. If, in a particular bank, a loan can belong to only one customer, and a customer can have several loans, then the relationship set from *customer* to *loan* is one to many. If a loan can belong to several customers (as can loans taken jointly by several business partners), the relationship set is many to many. Figure 2.3 depicts this type of relationship.

2.2.2 Participation Constraints

The participation of an entity set E in a relationship set R is said to be **total** if every entity in E participates in at least one relationship in R. If only some entities in E participate in relationships in R, the participation of entity set E in relationship R is said to be **partial**. For example, we expect every loan entity to be related to at least one customer through the *borrower* relationship. Therefore the participation of *loan* in

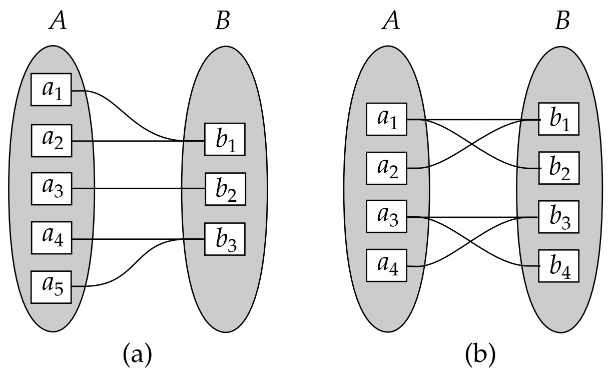


Figure 2.5 Mapping cardinalities. (a) Many to one. (b) Many to many.

the relationship set *borrower* is total. In contrast, an individual can be a bank customer whether or not she has a loan with the bank. Hence, it is possible that only some of the *customer* entities are related to the *loan* entity set through the *borrower* relationship, and the participation of *customer* in the *borrower* relationship set is therefore partial.

2.3 Keys

We must have a way to specify how entities within a given entity set are distinguished. Conceptually, individual entities are distinct; from a database perspective, however, the difference among them must be expressed in terms of their attributes.

Therefore, the values of the attribute values of an entity must be such that they can *uniquely identify* the entity. In other words, no two entities in an entity set are allowed to have exactly the same value for all attributes.

A *key* allows us to identify a set of attributes that suffice to distinguish entities from each other. Keys also help uniquely identify relationships, and thus distinguish relationships from each other.

2.3.1 Entity Sets

A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely an entity in the entity set. For example, the *customer-id* attribute of the entity set *customer* is sufficient to distinguish one *customer* entity from another. Thus, *customer-id* is a superkey. Similarly, the combination of *customer-name* and *customer-id* is a superkey for the entity set *customer*. The *customer-name* attribute of *customer* is not a superkey, because several people might have the same name.

The concept of a superkey is not sufficient for our purposes, since, as we saw, a superkey may contain extraneous attributes. If *K* is a superkey, then so is any superset of *K*. We are often interested in superkeys for which no proper subset is a superkey. Such minimal superkeys are called **candidate keys**.

It is possible that several distinct sets of attributes could serve as a candidate key. Suppose that a combination of *customer-name* and *customer-street* is sufficient to distinguish among members of the *customer* entity set. Then, both {*customer-id*} and {*customer-name*, *customer-street*} are candidate keys. Although the attributes *customer-id* and *customer-name* together can distinguish *customer* entities, their combination does not form a candidate key, since the attribute *customer-id* alone is a candidate key.

We shall use the term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying entities within an entity set. A key (primary, candidate, and super) is a property of the entity set, rather than of the individual entities. Any two individual entities in the set are prohibited from having the same value on the key attributes at the same time. The designation of a key represents a constraint in the real-world enterprise being modeled.

Candidate keys must be chosen with care. As we noted, the name of a person is obviously not sufficient, because there may be many people with the same name. In the United States, the social-security number attribute of a person would be a candidate key. Since non-U.S. residents usually do not have social-security numbers, international enterprises must generate their own unique identifiers. An alternative is to use some unique combination of other attributes as a key.

The primary key should be chosen such that its attributes are never, or very rarely, changed. For instance, the address field of a person should not be part of the primary key, since it is likely to change. Social-security numbers, on the other hand, are guaranteed to never change. Unique identifiers generated by enterprises generally do not change, except if two enterprises merge; in such a case the same identifier may have been issued by both enterprises, and a reallocation of identifiers may be required to make sure they are unique.

2.3.2 Relationship Sets

The primary key of an entity set allows us to distinguish among the various entities of the set. We need a similar mechanism to distinguish among the various relationships of a relationship set.

Let *R* be a relationship set involving entity sets $E_1,\ E_2, \ldots, E_n$. Let *primary-key*(E_i) denote the set of attributes that forms the primary key for entity set E_i. Assume for now that the attribute names of all primary keys are unique, and each entity set participates only once in the relationship. The composition of the primary key for a relationship set depends on the set of attributes associated with the relationship set *R*.

If the relationship set *R* has no attributes associated with it, then the set of attributes

$$\textit{primary-key}(E_1) \cup \textit{primary-key}(E_2) \cup \cdots \cup \textit{primary-key}(E_n)$$

describes an individual relationship in set R.

If the relationship set *R* has attributes $a_1, a_2, \cdots, a_m$ associated with it, then the set of attributes

$$\textit{primary-key}(E_1) \cup \textit{primary-key}(E_2) \cup \cdots \cup \textit{primary-key}(E_n) \ \cup \{a_1, a_2, \ldots, a_m\}$$

describes an individual relationship in set R.

In both of the above cases, the set of attributes

$$\textit{primary-key}(E_1) \cup \textit{primary-key}(E_2) \cup \cdots \cup \textit{primary-key}(E_n)$$

forms a superkey for the relationship set.

In case the attribute names of primary keys are not unique across entity sets, the attributes are renamed to distinguish them; the name of the entity set combined with the name of the attribute would form a unique name. In case an entity set participates more than once in a relationship set (as in the *works-for* relationship in Section 2.1.2), the role name is used instead of the name of the entity set, to form a unique attribute name.

The structure of the primary key for the relationship set depends on the mapping cardinality of the relationship set. As an illustration, consider the entity sets *customer* and *account*, and the relationship set *depositor*, with attribute *access-date*, in Section 2.1.2. Suppose that the relationship set is many to many. Then the primary key of *depositor* consists of the union of the primary keys of *customer* and *account*. However, if a customer can have only one account—that is, if the *depositor* relationship is many to one from *customer* to *account*—then the primary key of *depositor* is simply the primary key of *customer*. Similarly, if the relationship is many to one from *account* to *customer*—that is, each account is owned by at most one customer—then the primary key of *depositor* is simply the primary key of *account*. For one-to-one relationships either primary key can be used.

For nonbinary relationships, if no cardinality constraints are present then the superkey formed as described earlier in this section is the only candidate key, and it is chosen as the primary key. The choice of the primary key is more complicated if cardinality constraints are present. Since we have not discussed how to specify cardinality constraints on nonbinary relations, we do not discuss this issue further in this chapter. We consider the issue in more detail in Section 7.3.

2.4 Design 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.

2.4.1 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* permits employees 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 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.

2.4.2 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 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: (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 relationship 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.

2.4.3 Binary versus n-ary 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:

- R_A, relating E and A
- R_B, relating E and B
- R_C, 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 (a_i, b_i, c_i) in the relationship set R, we create a new entity e_i in the entity set E. Then, in each of the three new relationship sets, we insert a relationship as follows:

- (e_i, a_i) in R_A
- (e_i, b_i) in R_B
- (e_i, c_i) in R_C

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 R_A, R_B, and R_C.

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 Downtown; 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.

2.4.4 Placement of Relationship Attributes

The cardinality ratio of a relationship can affect the placement of relationship attributes. 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 instance of *customer*, making this attribute designation would have the same meaning

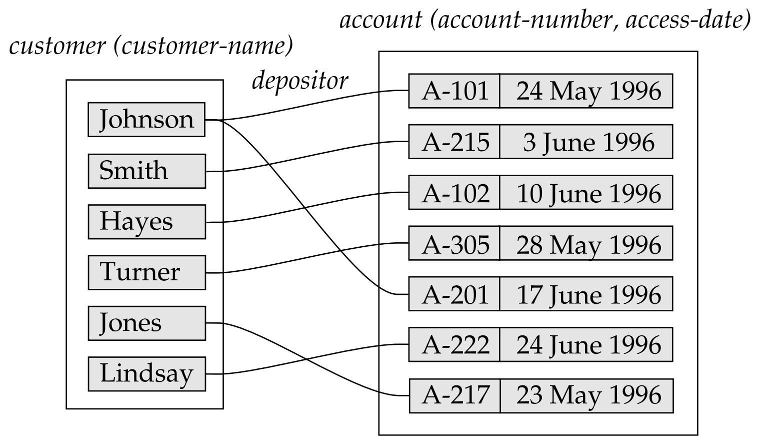


Figure 2.6 *Access-date* as attribute of the *account* entity set.

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.

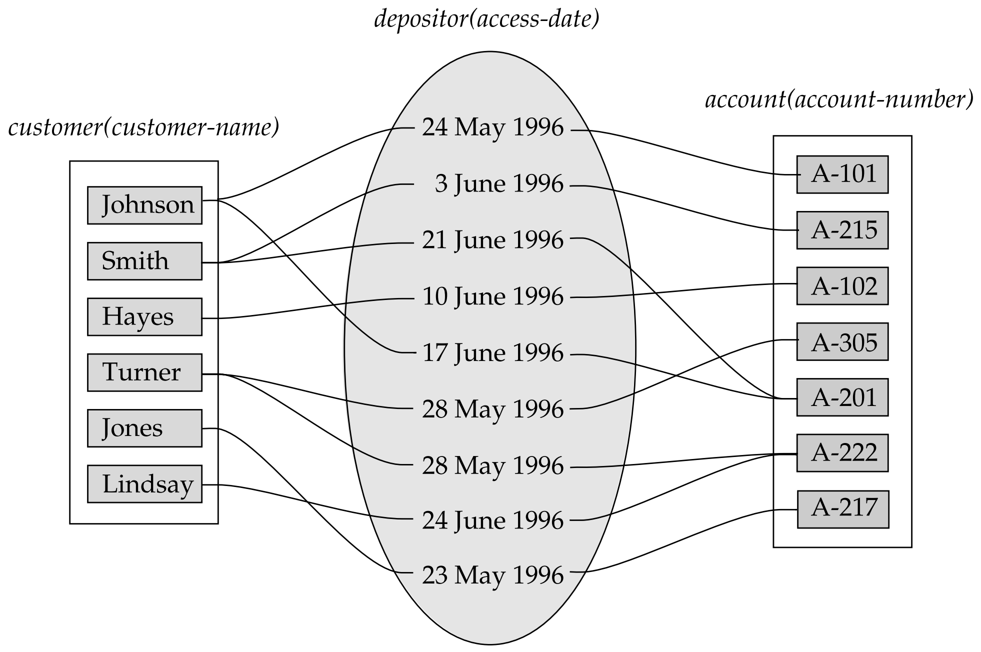


Figure 2.7 *Access-date* as attribute of the *depositor* relationship set.

2.5 Entity-Relationship Diagram

As we saw briefly in Section 1.4, an **E-R diagram** can express the overall logical structure of a database graphically. E-R diagrams are simple and clear—qualities that may well account in large part for the widespread use of the E-R model. Such a diagram consists of the following major components:

- **Rectangles**, which represent entity sets
- **Ellipses**, which represent attributes
- **Diamonds**, which represent relationship sets
- **Lines**, which link attributes to entity sets and entity sets to relationship sets
- **Double ellipses**, which represent multivalued attributes
- **Dashed ellipses**, which denote derived attributes
- **Double lines**, which indicate total participation of an entity in a relationship set
- **Double rectangles**, which represent weak entity sets (described later, in Section 2.6.)

Consider the entity-relationship diagram in Figure 2.8, which consists of two entity sets, *customer* and *loan*, related through a binary relationship set *borrower*. The attributes associated with *customer* are *customer-id*, *customer-name*, *customer-street*, and *customer-city*. The attributes associated with *loan* are *loan-number* and *amount*. In Figure 2.8, attributes of an entity set that are members of the primary key are underlined.

The relationship set *borrower* may be many-to-many, one-to-many, many-to-one, or one-to-one. To distinguish among these types, we draw either a directed line (→) or an undirected line (—) between the relationship set and the entity set in question.

- A directed line from the relationship set *borrower* to the entity set *loan* specifies that *borrower* is either a one-to-one or many-to-one relationship set, from *customer* to *loan*; *borrower* cannot be a many-to-many or a one-to-many relationship set from *customer* to *loan*.

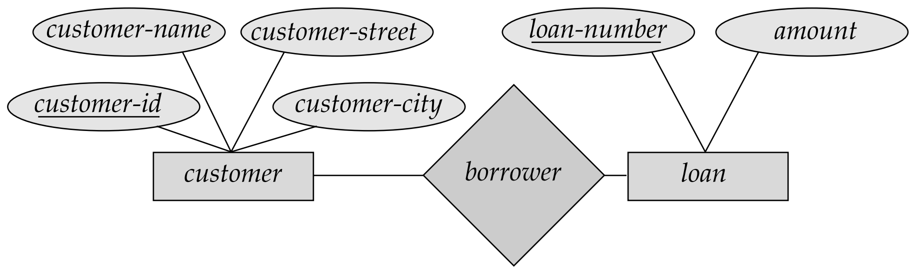


Figure 2.8 E-R diagram corresponding to customers and loans.

- An undirected line from the relationship set *borrower* to the entity set *loan* specifies that *borrower* is either a many-to-many or one-to-many relationship set from *customer* to *loan*.

Returning to the E-R diagram of Figure 2.8, we see that the relationship set *borrower* is many-to-many. If the relationship set *borrower* were one-to-many, from *customer* to *loan*, then the line from *borrower* to *customer* would be directed, with an arrow pointing to the *customer* entity set (Figure 2.9a). Similarly, if the relationship set *borrower* were many-to-one from *customer* to *loan*, then the line from *borrower* to *loan* would have an arrow pointing to the *loan* entity set (Figure 2.9b). Finally, if the relationship set *borrower* were one-to-one, then both lines from *borrower* would have arrows:

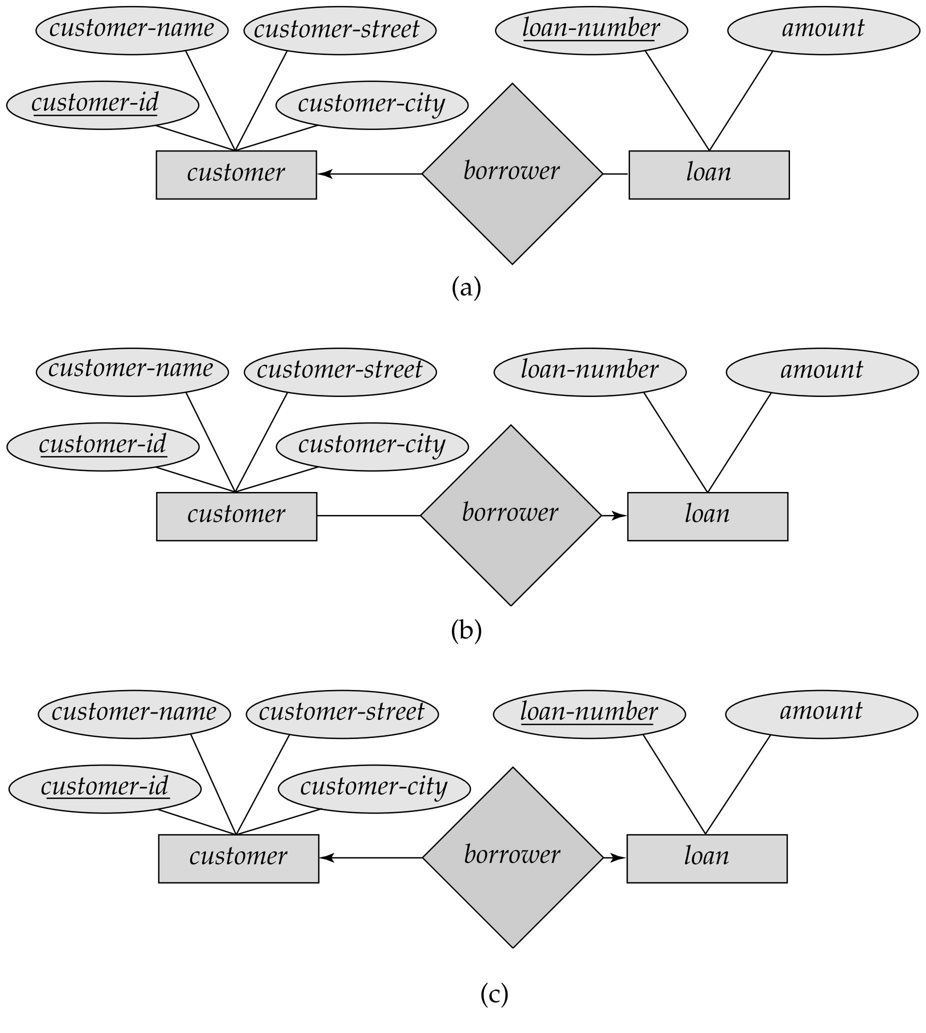


Figure 2.9 Relationships. (a) one to many. (b) many to one. (c) one-to-one.

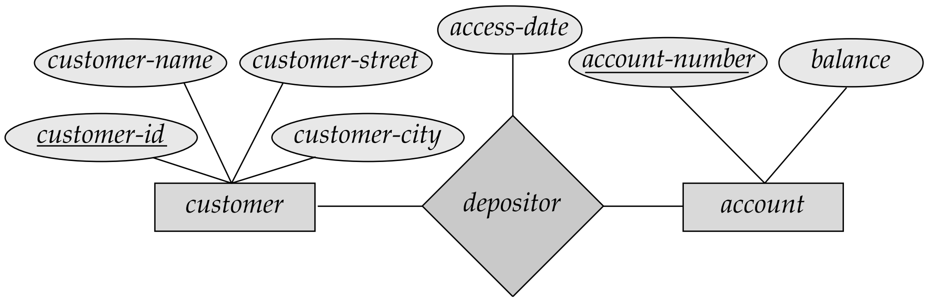


Figure 2.10 E-R diagram with an attribute attached to a relationship set.

one pointing to the *loan* entity set and one pointing to the *customer* entity set (Figure 2.9c).

If a relationship set has also some attributes associated with it, then we link these attributes to that relationship set. For example, in Figure 2.10, we have the *access-date* descriptive attribute attached to the relationship set *depositor* to specify the most recent date on which a customer accessed that account.

Figure 2.11 shows how composite attributes can be represented in the E-R notation. Here, a composite attribute *name*, with component attributes *first-name*, *middle-initial*, and *last-name* replaces the simple attribute *customer-name* of *customer*. Also, a composite attribute *address*, whose component attributes are *street*, *city*, *state*, and *zip-code* replaces the attributes *customer-street* and *customer-city* of *customer*. The attribute *street* is itself a composite attribute whose component attributes are *street-number*, *street-name*, and *apartment number*.

Figure 2.11 also illustrates a multivalued attribute *phone-number*, depicted by a double ellipse, and a derived attribute *age*, depicted by a dashed ellipse.

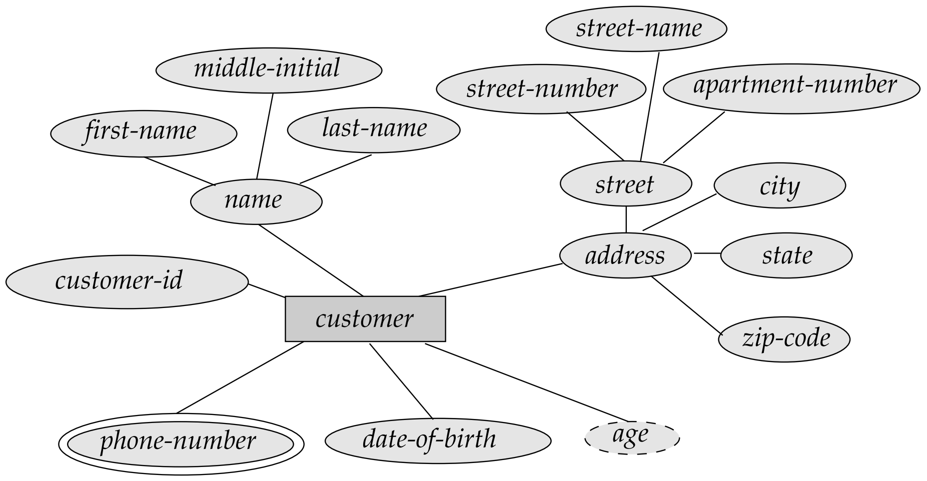


Figure 2.11 E-R diagram with composite, multivalued, and derived attributes.

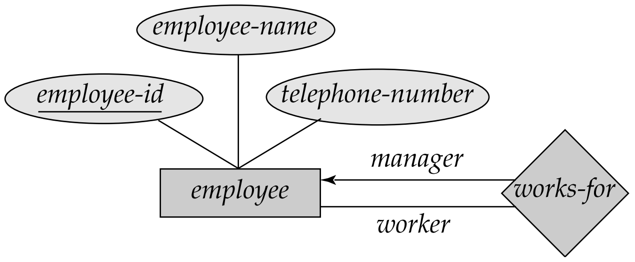


Figure 2.12 E-R diagram with role indicators.

We indicate roles in E-R diagrams by labeling the lines that connect diamonds to rectangles. Figure 2.12 shows the role indicators *manager* and *worker* between the *employee* entity set and the *works-for* relationship set.

Nonbinary relationship sets can be specified easily in an E-R diagram. Figure 2.13 consists of the three entity sets *employee*, *job*, and *branch*, related through the relationship set *works-on*.

We can specify some types of many-to-one relationships in the case of nonbinary relationship sets. Suppose an employee can have at most one job in each branch (for example, Jones cannot be a manager and an auditor at the same branch). This constraint can be specified by an arrow pointing to *job* on the edge from *works-on*.

We permit at most one arrow out of a relationship set, since an E-R diagram with two or more arrows out of a nonbinary relationship set can be interpreted in two ways. Suppose there is a relationship set R between entity sets $A_1, A_2, \ldots, A_n$, and the only arrows are on the edges to entity sets $A_{i+1}, A_{i+2}, \ldots, A_n$. Then, the two possible interpretations are:

1. A particular combination of entities from $A_1, A_2, \ldots, A_i$ can be associated with at most one combination of entities from $A_{i+1}, A_{i+2}, \ldots, A_n$. Thus, the primary key for the relationship R can be constructed by the union of the primary keys of $A_1, A_2, \ldots, A_i$.

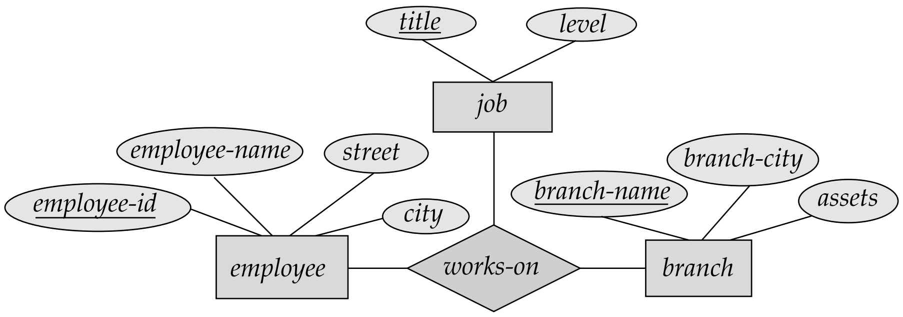


Figure 2.13 E-R diagram with a ternary relationship.

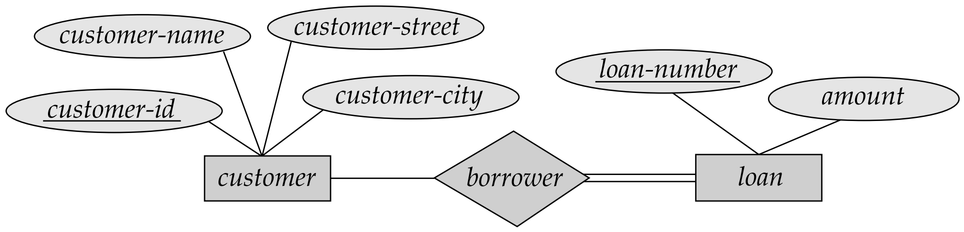


Figure 2.14 Total participation of an entity set in a relationship set.

2. For each entity set A_k, $i < k \leq n$, each combination of the entities from the other entity sets can be associated with at most one entity from A_k. Each set $\{A_1, A_2, \ldots, A_{k-1}, A_{k+1}, \ldots, A_n\}$, for $i < k \leq n$, then forms a candidate key.

Each of these interpretations has been used in different books and systems. To avoid confusion, we permit only one arrow out of a relationship set, in which case the two interpretations are equivalent. In Chapter 7 (Section 7.3) we study the notion of *functional dependencies*, which allow either of these interpretations to be specified in an unambiguous manner.

Double lines are used in an E-R diagram to indicate that the participation of an entity set in a relationship set is total; that is, each entity in the entity set occurs in at least one relationship in that relationship set. For instance, consider the relationship *borrower* between customers and loans. A double line from *loan* to *borrower*, as in Figure 2.14, indicates that each loan must have at least one associated customer.

E-R diagrams also provide a way to indicate more complex constraints on the number of times each entity participates in relationships in a relationship set. An edge between an entity set and a binary relationship set can have an associated minimum and maximum cardinality, shown in the form $l..h$, where l is the minimum and h the maximum cardinality. A minimum value of 1 indicates total participation of the entity set in the relationship set. A maximum value of 1 indicates that the entity participates in at most one relationship, while a maximum value $*$ indicates no limit. Note that a label $1..*$ on an edge is equivalent to a double line.

For example, consider Figure 2.15. The edge between *loan* and *borrower* has a cardinality constraint of $1..1$, meaning the minimum and the maximum cardinality are both 1. That is, each loan must have exactly one associated customer. The limit $0..*$ on the edge from *customer* to *borrower* indicates that a customer can have zero or more loans. Thus, the relationship *borrower* is one to many from *customer* to *loan*, and further the participation of *loan* in *borrower* is total.

It is easy to misinterpret the $0..*$ on the edge between *customer* and *borrower*, and think that the relationship *borrower* is many to one from *customer* to *loan*—this is exactly the reverse of the correct interpretation.

If both edges from a binary relationship have a maximum value of 1, the relationship is one to one. If we had specified a cardinality limit of $1..*$ on the edge between *customer* and *borrower*, we would be saying that each customer must have at least one loan.

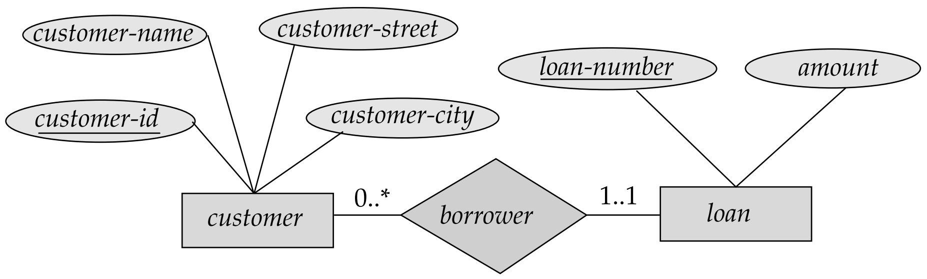


Figure 2.15 Cardinality limits on relationship sets.

2.6 Weak Entity Sets

An entity set may not have sufficient attributes to form a primary key. Such an entity set is termed a **weak entity set**. An entity set that has a primary key is termed a **strong entity set**.

As an illustration, consider the entity set *payment*, which has the three attributes: *payment-number*, *payment-date*, and *payment-amount*. Payment numbers are typically sequential numbers, starting from 1, generated separately for each loan. Thus, although each *payment* entity is distinct, payments for different loans may share the same payment number. Thus, this entity set does not have a primary key; it is a weak entity set.

For a weak entity set to be meaningful, it must be associated with another entity set, called the **identifying** or **owner entity set**. Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set. The identifying entity set is said to **own** the weak entity set that it identifies. The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**. The identifying relationship is many to one from the weak entity set to the identifying entity set, and the participation of the weak entity set in the relationship is total.

In our example, the identifying entity set for *payment* is *loan*, and a relationship *loan-payment* that associates *payment* entities with their corresponding *loan* entities is the identifying relationship.

Although a weak entity set does not have a primary key, we nevertheless need a means of distinguishing among all those entities in the weak entity set that depend on one particular strong entity. The **discriminator** of a weak entity set is a set of attributes that allows this distinction to be made. For example, the discriminator of the weak entity set *payment* is the attribute *payment-number*, since, for each loan, a payment number uniquely identifies one single payment for that loan. The discriminator of a weak entity set is also called the *partial key* of the entity set.

The primary key of a weak entity set is formed by the primary key of the identifying entity set, plus the weak entity set's discriminator. In the case of the entity set *payment*, its primary key is {*loan-number*, *payment-number*}, where *loan-number* is the primary key of the identifying entity set, namely *loan*, and *payment-number* distinguishes *payment* entities within the same loan.

The identifying relationship set should have no descriptive attributes, since any required attributes can be associated with the weak entity set (see the discussion of moving relationship-set attributes to participating entity sets in Section 2.2.1).

A weak entity set can participate in relationships other than the identifying relationship. For instance, the *payment* entity could participate in a relationship with the *account* entity set, identifying the account from which the payment was made. A weak entity set may participate as owner in an identifying relationship with another weak entity set. It is also possible to have a weak entity set with more than one identifying entity set. A particular weak entity would then be identified by a combination of entities, one from each identifying entity set. The primary key of the weak entity set would consist of the union of the primary keys of the identifying entity sets, plus the discriminator of the weak entity set.

In E-R diagrams, a doubly outlined box indicates a weak entity set, and a doubly outlined diamond indicates the corresponding identifying relationship. In Figure 2.16, the weak entity set *payment* depends on the strong entity set *loan* via the relationship set *loan-payment*.

The figure also illustrates the use of double lines to indicate *total participation*—the participation of the (weak) entity set *payment* in the relationship *loan-payment* is total, meaning that every payment must be related via *loan-payment* to some loan. Finally, the arrow from *loan-payment* to *loan* indicates that each payment is for a single loan. The discriminator of a weak entity set also is underlined, but with a dashed, rather than a solid, line.

In some cases, the database designer may choose to express a weak entity set as a multivalued composite attribute of the owner entity set. In our example, this alternative would require that the entity set *loan* have a multivalued, composite attribute *payment*, consisting of *payment-number*, *payment-date*, and *payment-amount*. A weak entity set may be more appropriately modeled as an attribute if it participates in only the identifying relationship, and if it has few attributes. Conversely, a weak-entity-set representation will more aptly model a situation where the set participates in relationships other than the identifying relationship, and where the weak entity set has several attributes.

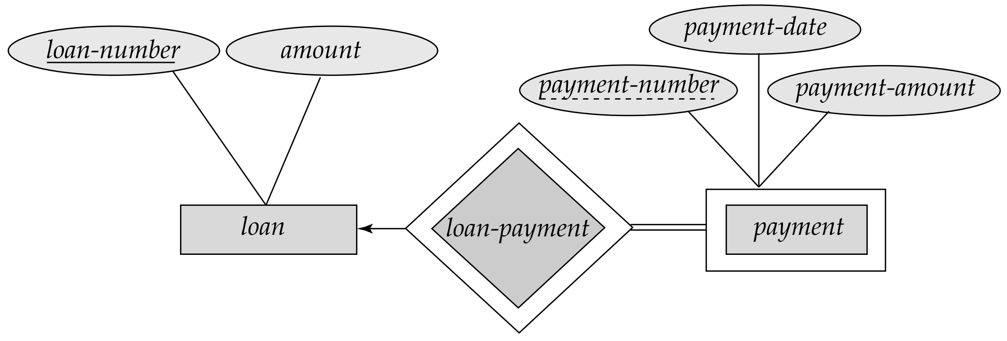


Figure 2.16 E-R diagram with a weak entity set.

As another example of an entity set that can be modeled as a weak entity set, consider offerings of a course at a university. The same course may be offered in different semesters, and within a semester there may be several sections for the same course. Thus we can create a weak entity set *course-offering*, existence dependent on *course*; different offerings of the same course are identified by a *semester* and a *section-number*, which form a discriminator but not a primary key.

2.7 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.

2.7.1 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 attributes of entity set *person* plus possibly additional attributes. For example, *customer* entities may be described further by the attribute *customer-id*, whereas *employee* entities 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.

2.7.2 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

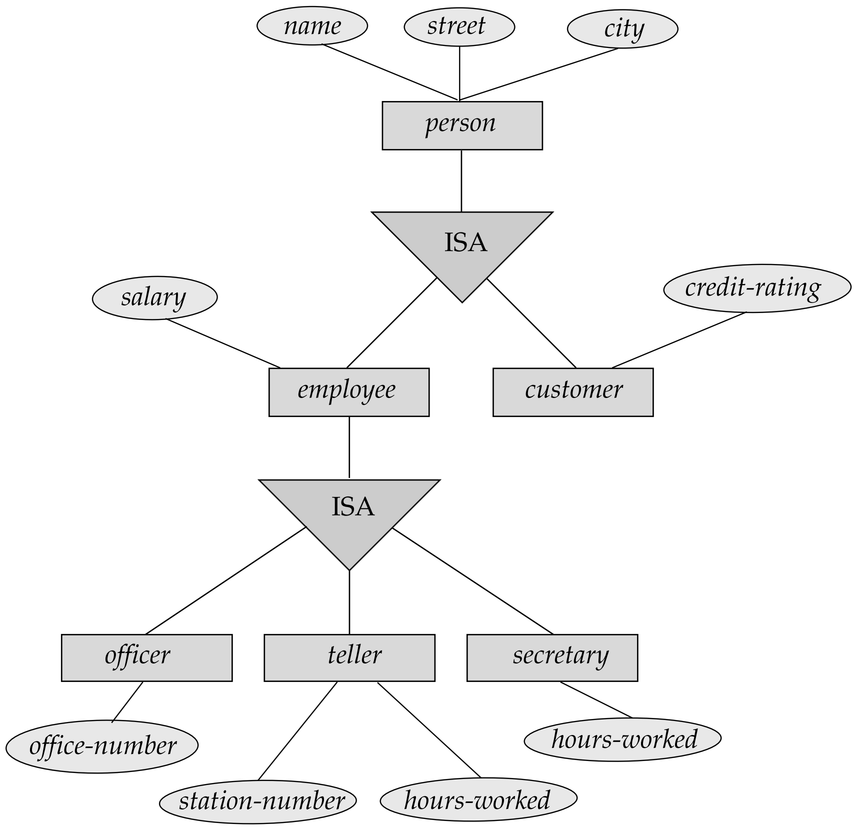


Figure 2.17 Specialization and generalization.

schema for an enterprise. In terms of the E-R diagram itself, we do not distinguish between 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.

2.7.3 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*.

2.7.4 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 *ac-*

count 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 entity 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 specialization, 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.

2.7.5 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 relationship *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.

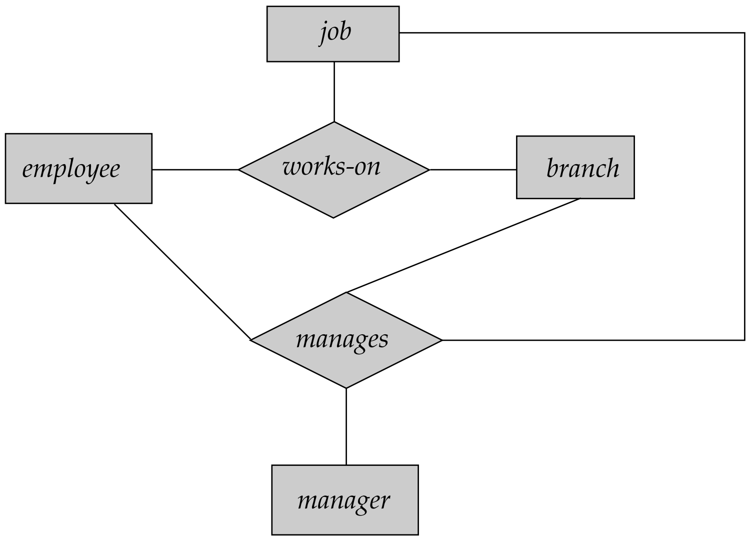


Figure 2.18 E-R diagram with redundant relationships.

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 attribute 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.

2.7.6 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.

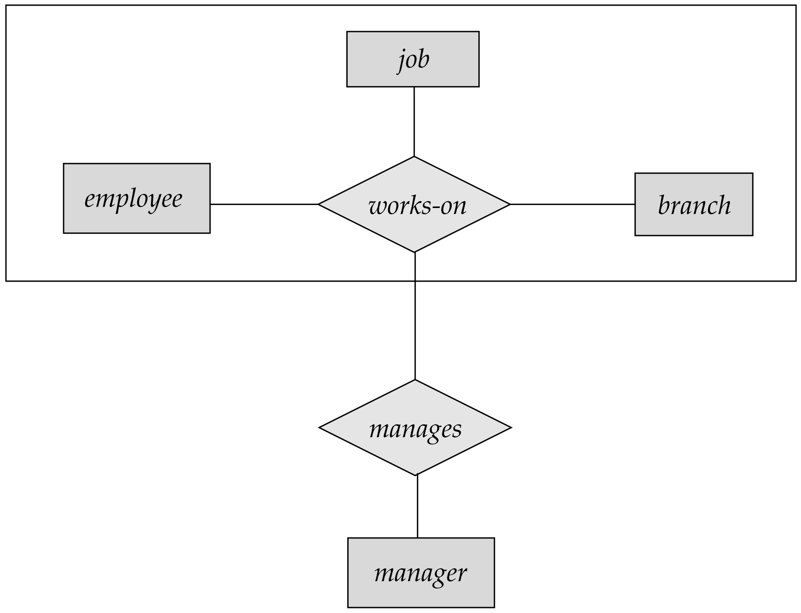


Figure 2.19 E-R diagram with aggregation.

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.

2.8 Design of an E-R Database Schema

The E-R data model gives us much flexibility in designing a database schema to model a given enterprise. In this section, we consider how a database designer may select from the wide range of alternatives. Among the designer's decisions are:

- Whether to use an attribute or an entity set to represent an object (discussed earlier in Section 2.2.1)
- Whether a real-world concept is expressed more accurately by an entity set or by a relationship set (Section 2.2.2)
- Whether to use a ternary relationship or a pair of binary relationships (Section 2.2.3)

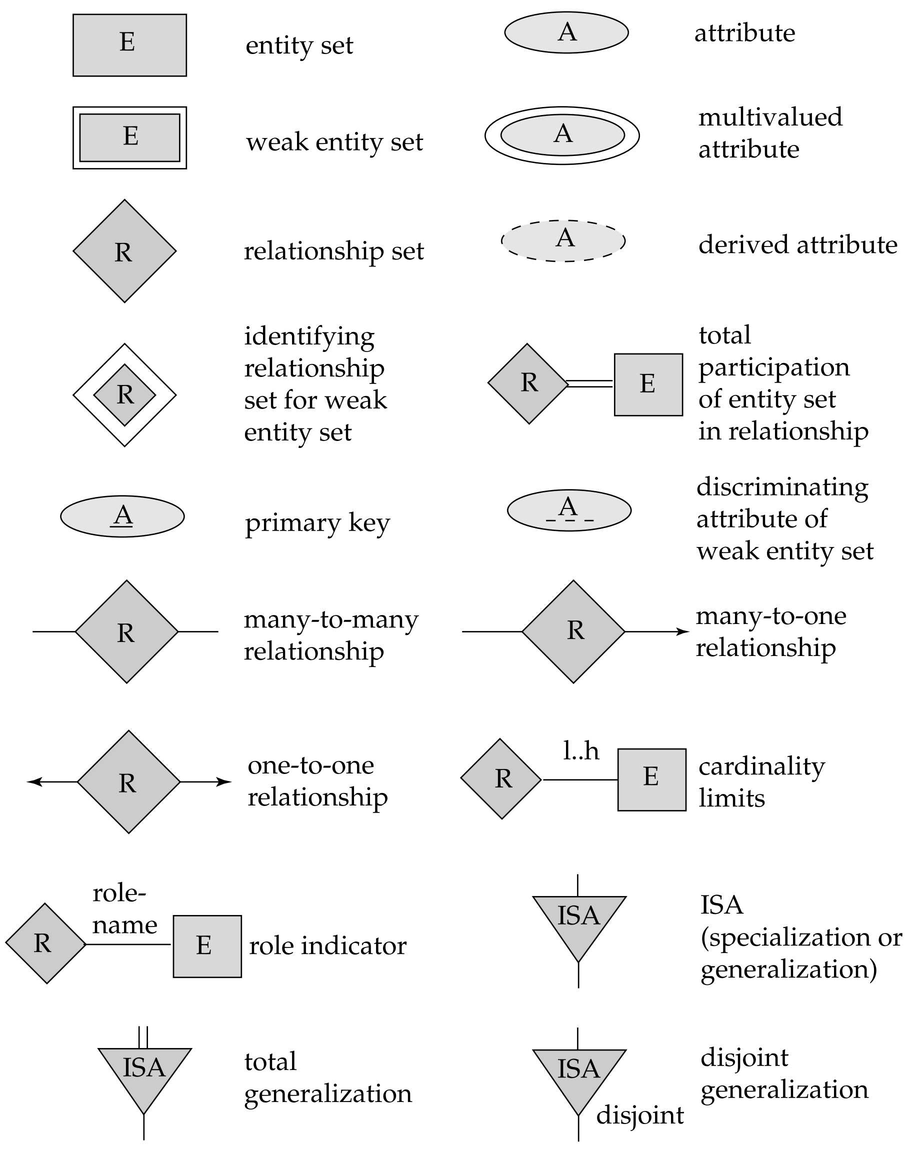


Figure 2.20 Symbols used in the E-R notation.

- Whether to use a strong or a weak entity set (Section 2.6); a strong entity set and its dependent weak entity sets may be regarded as a single "object" in the database, since weak entities are existence dependent on a strong entity
- Whether using generalization (Section 2.7.2) is appropriate; generalization, or a hierarchy of ISA relationships, contributes to modularity by allowing com-

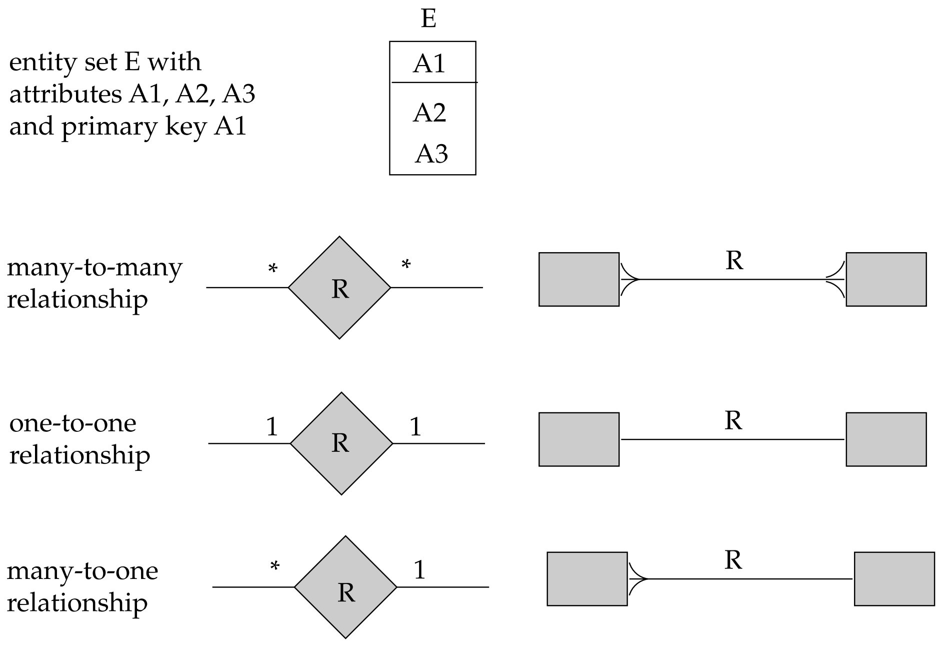

E
A1
A2
A3
entity set E with attributes A1, A2, A3 and primary key A1
many-to-many relationship
*
R
*
R
one-to-one relationship
1
R
1
R
many-to-one relationship
*
R
1
R

Figure 2.21 Alternative E-R notations.

mon attributes of similar entity sets to be represented in one place in an E-R diagram

- Whether using aggregation (Section 2.7.5) is appropriate; aggregation groups a part of an E-R diagram into a single entity set, allowing us to treat the aggregate entity set as a single unit without concern for the details of its internal structure.

We shall see that the database designer needs a good understanding of the enterprise being modeled to make these decisions.

2.8.1 Design Phases

A high-level data model serves the database designer by providing a conceptual framework in which to specify, in a systematic fashion, what the data requirements of the database users are, and how the database will be structured to fulfill these requirements. The initial phase of database design, then, is to characterize fully the data needs of the prospective database users. The database designer needs to interact extensively with domain experts and users to carry out this task. The outcome of this phase is a specification of user requirements.

Next, the designer chooses a data model, and by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database. The schema developed at this **conceptual-design** phase provides a detailed overview of the enterprise. Since we have studied only the E-R model so far, we shall use it to develop the conceptual schema. Stated in terms of the E-R model, the schema specifies all entity sets, relationship sets, attributes, and mapping constraints. The designer reviews the schema to confirm that all data requirements are indeed satisfied and are not in conflict with one another. She can also examine the design to remove any redundant features. Her focus at this point is describing the data and their relationships, rather than on specifying physical storage details.

A fully developed conceptual schema will also indicate the functional requirements of the enterprise. In a **specification of functional requirements**, users describe the kinds of operations (or transactions) that will be performed on the data. Example operations include modifying or updating data, searching for and retrieving specific data, and deleting data. At this stage of conceptual design, the designer can review the schema to ensure it meets functional requirements.

The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases. In the **logical-design phase**, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used. The designer uses the resulting system-specific database schema in the subsequent **physical-design phase**, in which the physical features of the database are specified. These features include the form of file organization and the internal storage structures; they are discussed in Chapter 11.

In this chapter, we cover only the concepts of the E-R model as used in the conceptual-schema-design phase. We have presented a brief overview of the database-design process to provide a context for the discussion of the E-R data model. Database design receives a full treatment in Chapter 7.

In Section 2.8.2, we apply the two initial database-design phases to our banking-enterprise example. We employ the E-R data model to translate user requirements into a conceptual design schema that is depicted as an E-R diagram.

2.8.2 Database Design for Banking Enterprise

We now look at the database-design requirements of a banking enterprise in more detail, and develop a more realistic, but also more complicated, design than what we have seen in our earlier examples. However, we do not attempt to model every aspect of the database-design for a bank; we consider only a few aspects, in order to illustrate the process of database design.

2.8.2.1 Data Requirements

The initial specification of user requirements may be based on interviews with the database users, and on the designer's own analysis of the enterprise. The description that arises from this design phase serves as the basis for specifying the conceptual structure of the database. Here are the major characteristics of the banking enterprise.

- The bank is organized into branches. Each branch is located in a particular city and is identified by a unique name. The bank monitors the assets of each branch.

- Bank customers are identified by their *customer-id* values. The bank stores each customer's name, and the street and city where the customer lives. Customers may have accounts and can take out loans. A customer may be associated with a particular banker, who may act as a loan officer or personal banker for that customer.
- Bank employees are identified by their *employee-id* values. The bank administration stores the name and telephone number of each employee, the names of the employee's dependents, and the *employee-id* number of the employee's manager. The bank also keeps track of the employee's start date and, thus, length of employment.
- The bank offers two types of accounts—savings and checking accounts. Accounts can be held by more than one customer, and a customer can have more than one account. Each account is assigned a unique account number. The bank maintains a record of each account's balance, and the most recent date on which the account was accessed by each customer holding the account. In addition, each savings account has an interest rate, and overdrafts are recorded for each checking account.
- A loan originates at a particular branch and can be held by one or more customers. A loan is identified by a unique loan number. For each loan, the bank keeps track of the loan amount and the loan payments. Although a loan-payment number does not uniquely identify a particular payment among those for all the bank's loans, a payment number does identify a particular payment for a specific loan. The date and amount are recorded for each payment.

In a real banking enterprise, the bank would keep track of deposits and withdrawals from savings and checking accounts, just as it keeps track of payments to loan accounts. Since the modeling requirements for that tracking are similar, and we would like to keep our example application small, we do not keep track of such deposits and withdrawals in our model.

2.8.2.2 Entity Sets Designation

Our specification of data requirements serves as the starting point for constructing a conceptual schema for the database. From the characteristics listed in Section 2.8.2.1, we begin to identify entity sets and their attributes:

- The *branch* entity set, with attributes *branch-name*, *branch-city*, and *assets*.
- The *customer* entity set, with attributes *customer-id*, *customer-name*, *customer-street*; and *customer-city*. A possible additional attribute is *banker-name*.
- The *employee* entity set, with attributes *employee-id*, *employee-name*, *telephone-number*, *salary*, and *manager*. Additional descriptive features are the multivalued attribute *dependent-name*, the base attribute *start-date*, and the derived attribute *employment-length*.

- Two account entity sets—*savings-account* and *checking-account*—with the common attributes of *account-number* and *balance*; in addition, *savings-account* has the attribute *interest-rate* and *checking-account* has the attribute *overdraft-amount*.
- The *loan* entity set, with the attributes *loan-number*, *amount*, and *originating-branch*.
- The weak entity set *loan-payment*, with attributes *payment-number*, *payment-date*, and *payment-amount*.

2.8.2.3 Relationship Sets Designation

We now return to the rudimentary design scheme of Section 2.8.2.2 and specify the following relationship sets and mapping cardinalities. In the process, we also refine some of the decisions we made earlier regarding attributes of entity sets.

- *borrower*, a many-to-many relationship set between *customer* and *loan*.
- *loan-branch*, a many-to-one relationship set that indicates in which branch a loan originated. Note that this relationship set replaces the attribute *originating-branch* of the entity set *loan*.
- *loan-payment*, a one-to-many relationship from *loan* to *payment*, which documents that a payment is made on a loan.
- *depositor*, with relationship attribute *access-date*, a many-to-many relationship set between *customer* and *account*, indicating that a customer owns an account.
- *cust-banker*, with relationship attribute *type*, a many-to-one relationship set expressing that a customer can be advised by a bank employee, and that a bank employee can advise one or more customers. Note that this relationship set has replaced the attribute *banker-name* of the entity set *customer*.
- *works-for*, a relationship set between *employee* entities with role indicators *manager* and *worker*; the mapping cardinalities express that an employee works for only one manager and that a manager supervises one or more employees. Note that this relationship set has replaced the *manager* attribute of *employee*.

2.8.2.4 E-R Diagram

Drawing on the discussions in Section 2.8.2.3, we now present the completed E-R diagram for our example banking enterprise. Figure 2.22 depicts the full representation of a conceptual model of a bank, expressed in terms of E-R concepts. The diagram includes the entity sets, attributes, relationship sets, and mapping cardinalities arrived at through the design processes of Sections 2.8.2.1 and 2.8.2.2, and refined in Section 2.8.2.3.

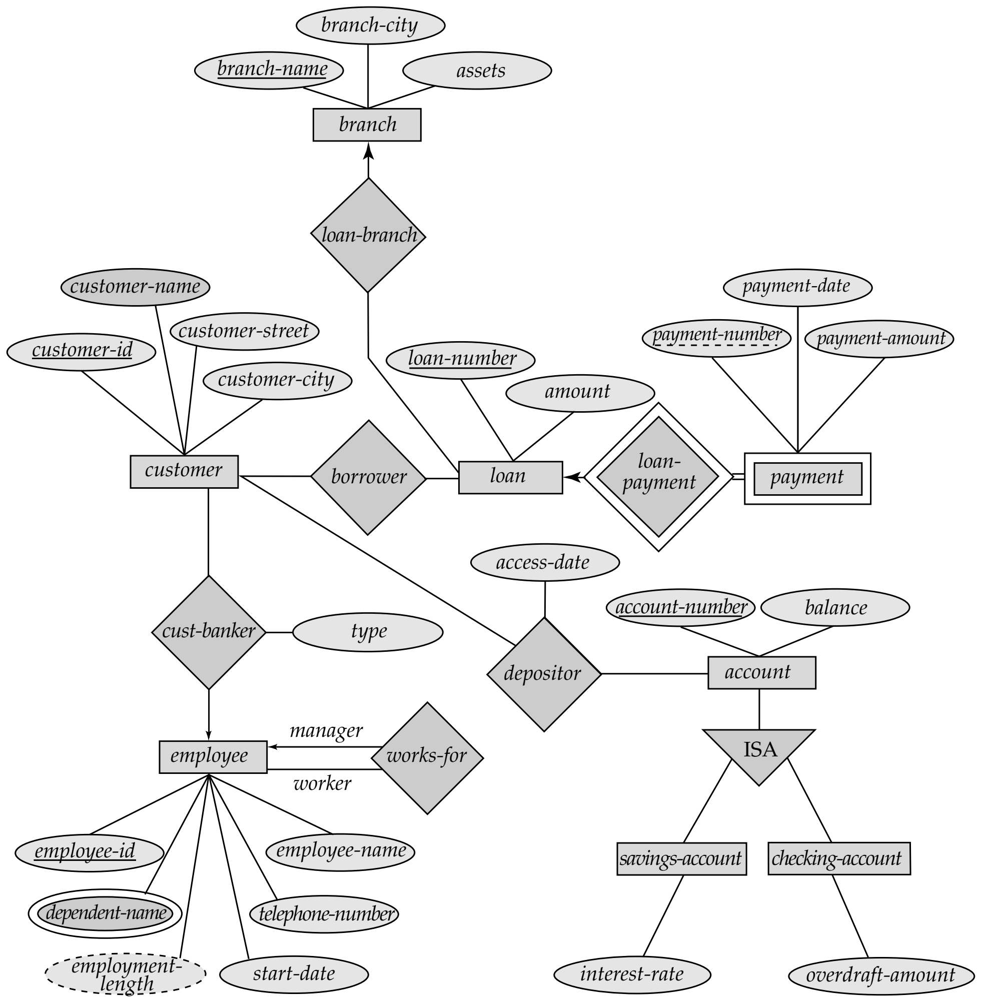


Figure 2.22 E-R diagram for a banking enterprise.

2.9 Reduction of an E-R Schema to Tables

We can represent a database that conforms to an E-R database schema by a collection of tables. For each entity set and for each relationship set in the database, there is a unique table to which we assign the name of the corresponding entity set or relationship set. Each table has multiple columns, each of which has a unique name.

Both the E-R model and the relational-database model are abstract, logical representations of real-world enterprises. Because the two models employ similar design principles, we can convert an E-R design into a relational design. Converting a database representation from an E-R diagram to a table format is the way we arrive at a relational-database design from an E-R diagram. Although important differences exist between a relation and a table, informally, a relation can be considered to be a table of values.

In this section, we describe how an E-R schema can be represented by tables; and in Chapter 3, we show how to generate a relational-database schema from an E-R schema.

The constraints specified in an E-R diagram, such as primary keys and cardinality constraints, are mapped to constraints on the tables generated from the E-R diagram. We provide more details about this mapping in Chapter 6 after describing how to specify constraints on tables.

2.9.1 Tabular Representation of Strong Entity Sets

Let *E* be a strong entity set with descriptive attributes $a_1, \ a_2, \ldots, a_n$. We represent this entity by a table called *E* with *n* distinct columns, each of which corresponds to one of the attributes of *E*. Each row in this table corresponds to one entity of the entity set *E*.

As an illustration, consider the entity set *loan* of the E-R diagram in Figure 2.8. This entity set has two attributes: *loan-number* and *amount*. We represent this entity set by a table called *loan*, with two columns, as in Figure 2.23. The row

(L-17, 1000)

in the *loan* table means that loan number L-17 has a loan amount of $1000. We can add a new entity to the database by inserting a row into a table. We can also delete or modify rows.

Let D_1 denote the set of all loan numbers, and let D_2 denote the set of all balances. Any row of the *loan* table must consist of a 2-tuple $(v_1, \ v_2)$, where v_1 is a loan (that is, v_1 is in set D_1) and v_2 is an amount (that is, v_2 is in set D_2). In general, the *loan* table will contain only a subset of the set of all possible rows. We refer to the set of all possible rows of *loan* as the *Cartesian product* of D_1 and D_2, denoted by

$$D_1 \ \times \ D_2$$

In general, if we have a table of *n* columns, we denote the Cartesian product of $D_1, \ D_2, \cdots, D_n$ by

$$D_1 \ \times \ D_2 \ \times \ \cdots \ \times \ D_{n-1} \ \times \ D_n$$

loan-number	*amount*
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

Figure 2.23 The *loan* table.

customer-id	*customer-name*	*customer-street*	*customer-city*
019-28-3746	Smith	North	Rye
182-73-6091	Turner	Putnam	Stamford
192-83-7465	Johnson	Alma	Palo Alto
244-66-8800	Curry	North	Rye
321-12-3123	Jones	Main	Harrison
335-57-7991	Adams	Spring	Pittsfield
336-66-9999	Lindsay	Park	Pittsfield
677-89-9011	Hayes	Main	Harrison
963-96-3963	Williams	Nassau	Princeton

Figure 2.24 The *customer* table.

As another example, consider the entity set *customer* of the E-R diagram in Figure 2.8. This entity set has the attributes *customer-id*, *customer-name*, *customer-street*, and *customer-city*. The table corresponding to *customer* has four columns, as in Figure 2.24.

2.9.2 Tabular Representation of Weak Entity Sets

Let A be a weak entity set with attributes $a_1, a_2, \ldots, a_m$. Let B be the strong entity set on which A depends. Let the primary key of B consist of attributes $b_1,\ b_2, \ldots, b_n$. We represent the entity set A by a table called A with one column for each attribute of the set:

$$\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$$

As an illustration, consider the entity set *payment* in the E-R diagram of Figure 2.16. This entity set has three attributes: *payment-number*, *payment-date*, and *payment-amount*. The primary key of the *loan* entity set, on which *payment* depends, is *loan-number*. Thus, we represent *payment* by a table with four columns labeled *loan-number*, *payment-number*, *payment-date*, and *payment-amount*, as in Figure 2.25.

2.9.3 Tabular Representation of Relationship Sets

Let R be a relationship set, let $a_1, a_2, \ldots, a_m$ be the set of attributes formed by the union of the primary keys of each of the entity sets participating in R, and let the descriptive attributes (if any) of R be $b_1, b_2, \ldots, b_n$. We represent this relationship set by a table called R with one column for each attribute of the set:

$$\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$$

As an illustration, consider the relationship set *borrower* in the E-R diagram of Figure 2.8. This relationship set involves the following two entity sets:

- *customer*, with the primary key *customer-id*
- *loan*, with the primary key *loan-number*

loan-number	*payment-number*	*payment-date*	*payment-amount*
L-11	53	7 June 2001	125
L-14	69	28 May 2001	500
L-15	22	23 May 2001	300
L-16	58	18 June 2001	135
L-17	5	10 May 2001	50
L-17	6	7 June 2001	50
L-17	7	17 June 2001	100
L-23	11	17 May 2001	75
L-93	103	3 June 2001	900
L-93	104	13 June 2001	200

Figure 2.25 The *payment* table.

Since the relationship set has no attributes, the *borrower* table has two columns, labeled *customer-id* and *loan-number*, as shown in Figure 2.26.

2.9.3.1 Redundancy of Tables

A relationship set linking a weak entity set to the corresponding strong entity set is treated specially. As we noted in Section 2.6, these relationships are many-to-one and have no descriptive attributes. Furthermore, the primary key of a weak entity set includes the primary key of the strong entity set. In the E-R diagram of Figure 2.16, the weak entity set *payment* is dependent on the strong entity set *loan* via the relationship set *loan-payment*. The primary key of *payment* is {*loan-number*, *payment-number*}, and the primary key of *loan* is {*loan-number*}. Since *loan-payment* has no descriptive attributes, the *loan-payment* table would have two columns, *loan-number* and *payment-number*. The table for the entity set *payment* has four columns, *loan-number*, *payment-number*, *payment-date*, and *payment-amount*. Every (*loan-number*, *payment-number*) combination in *loan-payment* would also be present in the *payment* table, and vice versa. Thus, the *loan-payment* table is redundant. In general, the table for the relationship set

customer-id	*loan-number*
019-28-3746	L-11
019-28-3746	L-23
244-66-8800	L-93
321-12-3123	L-17
335-57-7991	L-16
555-55-5555	L-14
677-89-9011	L-15
963-96-3963	L-17

Figure 2.26 The *borrower* table.

linking a weak entity set to its corresponding strong entity set is redundant and does not need to be present in a tabular representation of an E-R diagram.

2.9.3.2 Combination of Tables

Consider a many-to-one relationship set *AB* from entity set *A* to entity set *B*. Using our table-construction scheme outlined previously, we get three tables: *A*, *B*, and *AB*. Suppose further that the participation of *A* in the relationship is total; that is, every entity *a* in the entity set *A* must participate in the relationship *AB*. Then we can combine the tables *A* and *AB* to form a single table consisting of the union of columns of both tables.

As an illustration, consider the E-R diagram of Figure 2.27. The double line in the E-R diagram indicates that the participation of *account* in the *account-branch* is total. Hence, an account cannot exist without being associated with a particular branch. Further, the relationship set *account-branch* is many to one from *account* to *branch*. Therefore, we can combine the table for *account-branch* with the table for *account* and require only the following two tables:

- *account*, with attributes *account-number*, *balance*, and *branch-name*
- *branch*, with attributes *branch-name*, *branch-city*, and *assets*

2.9.4 Composite Attributes

We handle composite attributes by creating a separate attribute for each of the component attributes; we do not create a separate column for the composite attribute itself. Suppose *address* is a composite attribute of entity set *customer*, and the components of *address* are *street* and *city*. The table generated from *customer* would then contain columns *address-street* and *address-city*; there is no separate column for *address*.

2.9.5 Multivalued Attributes

We have seen that attributes in an E-R diagram generally map directly into columns for the appropriate tables. Multivalued attributes, however, are an exception; new tables are created for these attributes.

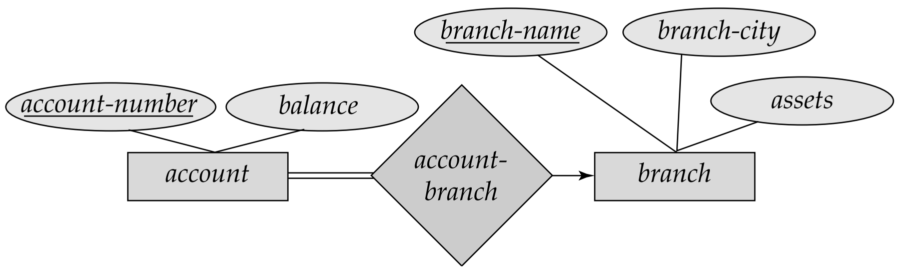


Figure 2.27 E-R diagram.

For a multivalued attribute *M*, we create a table *T* with a column *C* that corresponds to *M* and columns corresponding to the primary key of the entity set or relationship set of which *M* is an attribute. As an illustration, consider the E-R diagram in Figure 2.22. The diagram includes the multivalued attribute *dependent-name*. For this multivalued attribute, we create a table *dependent-name*, with columns *dname*, referring to the *dependent-name* attribute of *employee*, and *employee-id*, representing the primary key of the entity set *employee*. Each dependent of an employee is represented as a unique row in the table.

2.9.6 Tabular Representation of Generalization

There are two different methods for transforming to a tabular form an E-R diagram that includes generalization. Although we refer to the generalization in Figure 2.17 in this discussion, we simplify it by including only the first tier of lower-level entity sets—that is, *savings-account* and *checking-account*.

1. Create a table for the higher-level entity set. For each lower-level entity set, create a table that includes a column for each of the attributes of that entity set plus a column for each attribute of the primary key of the higher-level entity set. Thus, for the E-R diagram of Figure 2.17, we have three tables:
   - *account*, with attributes *account-number* and *balance*
   - *savings-account*, with attributes *account-number* and *interest-rate*
   - *checking-account*, with attributes *account-number* and *overdraft-amount*
2. An alternative representation is possible, if the generalization is disjoint and complete—that is, if no entity is a member of two lower-level entity sets directly below a higher-level entity set, and if every entity in the higher level entity set is also a member of one of the lower-level entity sets. Here, do not create a table for the higher-level entity set. Instead, for each lower-level entity set, create a table that includes a column for each of the attributes of that entity set plus a column for *each* attribute of the higher-level entity set. Then, for the E-R diagram of Figure 2.17, we have two tables.
   - *savings-account*, with attributes *account-number*, *balance*, and *interest-rate*
   - *checking-account*, with attributes *account-number*, *balance*, and *overdraft-amount*

   The *savings-account* and *checking-account* relations corresponding to these tables both have *account-number* as the primary key.

If the second method were used for an overlapping generalization, some values such as *balance* would be stored twice unnecessarily. Similarly, if the generalization were not complete—that is, if some accounts were neither savings nor checking accounts—then such accounts could not be represented with the second method.

2.9.7 Tabular Representation of Aggregation

Transforming an E-R diagram containing aggregation to a tabular form is straightforward. Consider the diagram of Figure 2.19. The table for the relationship set *manages* between the aggregation of *works-on* and the entity set *manager* includes a column for each attribute in the primary keys of the entity set *manager* and the relationship set *works-on*. It would also include a column for any descriptive attributes, if they exist, of the relationship set *manages*. We then transform the relationship sets and entity sets within the aggregated entity.

2.10 The Unified Modeling Language UML∗∗

Entity-relationship diagrams help model the data representation component of a software system. Data representation, however, forms only one part of an overall system design. Other components include models of user interactions with the system, specification of functional modules of the system and their interaction, etc. The **Unified Modeling Language** (UML), is a proposed standard for creating specifications of various components of a software system. Some of the parts of UML are:

- **Class diagram**. A class diagram is similar to an E-R diagram. Later in this section we illustrate a few features of class diagrams and how they relate to E-R diagrams.
- **Use case diagram**. Use case diagrams show the interaction between users and the system, in particular the steps of tasks that users perform (such as withdrawing money or registering for a course).
- **Activity diagram**. Activity diagrams depict the flow of tasks between various components of a system.
- **Implementation diagram**. Implementation diagrams show the system components and their interconnections, both at the software component level and the hardware component level.

We do not attempt to provide detailed coverage of the different parts of UML here. See the bibliographic notes for references on UML. Instead we illustrate some features of UML through examples.

Figure 2.28 shows several E-R diagram constructs and their equivalent UML class diagram constructs. We describe these constructs below. UML shows entity sets as boxes and, unlike E-R, shows attributes within the box rather than as separate ellipses. UML actually models objects, whereas E-R models entities. Objects are like entities, and have attributes, but additionally provide a set of functions (called methods) that can be invoked to compute values on the basis of attributes of the objects, or to update the object itself. Class diagrams can depict methods in addition to attributes. We cover objects in Chapter 8.

We represent binary relationship sets in UML by just drawing a line connecting the entity sets. We write the relationship set name adjacent to the line. We may also specify the role played by an entity set in a relationship set by writing the role name on the line, adjacent to the entity set. Alternatively, we may write the relationship set name in a box, along with attributes of the relationship set, and connect the box by a dotted line to the line depicting the relationship set. This box can then be treated as

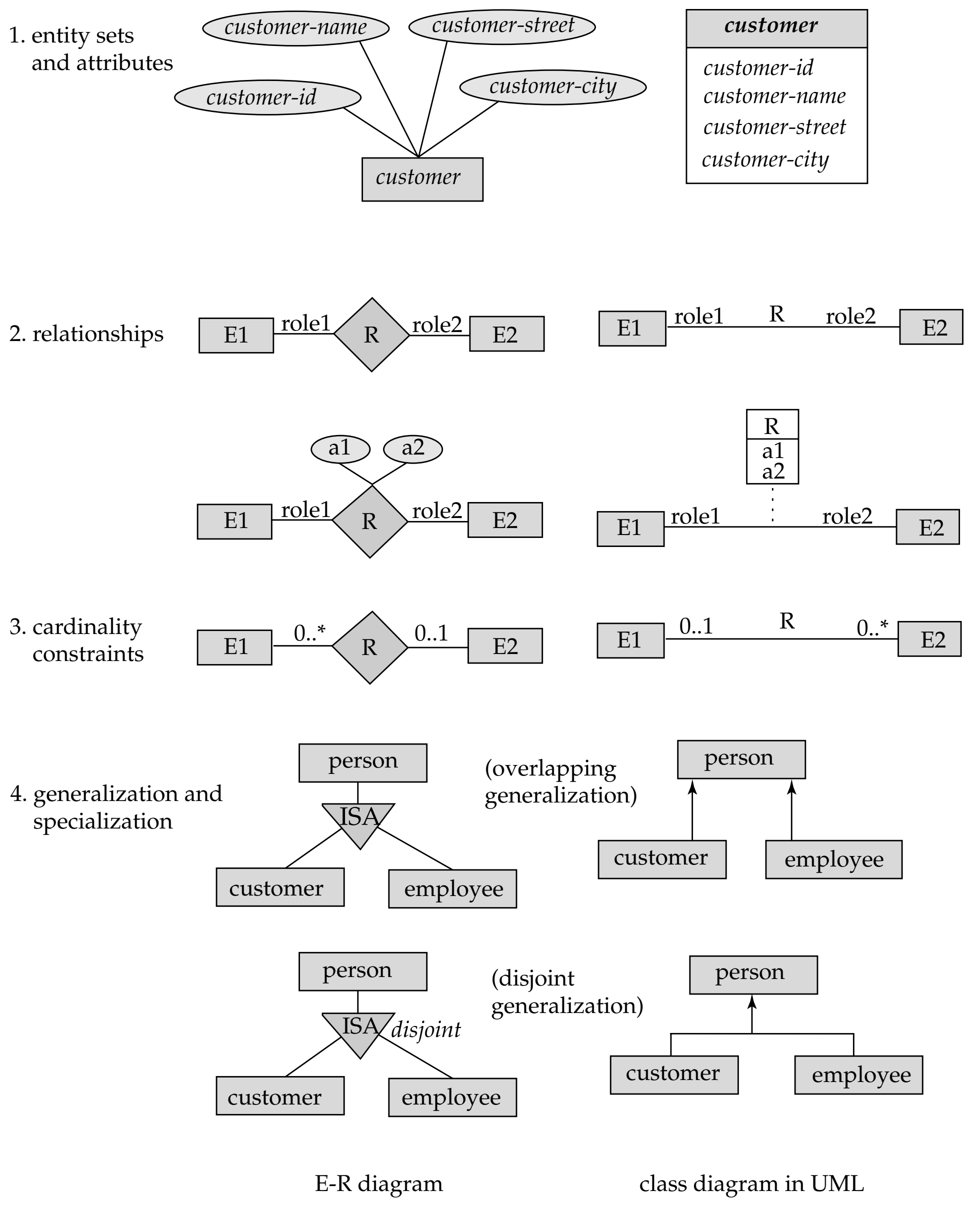


Figure 2.28 Symbols used in the UML class diagram notation.

an entity set, in the same way as an aggregation in E-R diagrams and can participate in relationships with other entity sets.

Nonbinary relationships cannot be directly represented in UML—they have to be converted to binary relationships by the technique we have seen earlier in Section 2.4.3.

Cardinality constraints are specified in UML in the same way as in E-R diagrams, in the form $l..h$, where l denotes the minimum and h the maximum number of relationships an entity can participate in. However, you should be aware that the positioning of the constraints is exactly the reverse of the positioning of constraints in E-R diagrams, as shown in Figure 2.28. The constraint $0..*$ on the $E2$ side and $0..1$ on the $E1$ side means that each $E2$ entity can participate in at most one relationship, whereas each $E1$ entity can participate in many relationships; in other words, the relationship is many to one from $E2$ to $E1$.

Single values such as 1 or $*$ may be written on edges; the single value 1 on an edge is treated as equivalent to $1..1$, while $*$ is equivalent to $0..*$.

We represent generalization and specialization in UML by connecting entity sets by a line with a triangle at the end corresponding to the more general entity set. For instance, the entity set *person* is a generalization of *customer* and *employee*. UML diagrams can also represent explicitly the constraints of disjoint/overlapping on generalizations. Figure 2.28 shows disjoint and overlapping generalizations of *customer* and *employee* to *person*. Recall that if the *customer*/*employee* to *person* generalization is disjoint, it means that no one can be both a *customer* and an *employee*. An overlapping generalization allows a person to be both a *customer* and an *employee*.

2.11 Summary

- The **entity-relationship (E-R)** data model is based on a perception of a real world that consists of a set of basic objects called **entities**, and of **relationships** among these objects.
- The model is intended primarily for the database-design process. It was developed to facilitate database design by allowing the specification of an **enterprise schema**. Such a schema represents the overall logical structure of the database. This overall structure can be expressed graphically by an **E-R diagram**.
- An **entity** is an object that exists in the real world and is distinguishable from other objects. We express the distinction by associating with each entity a set of attributes that describes the object.
- A **relationship** is an association among several entities. The collection of all entities of the same type is an **entity set**, and the collection of all relationships of the same type is a **relationship set**.
- **Mapping cardinalities** express the number of entities to which another entity can be associated via a relationship set.
- A **superkey** of an entity set is a set of one or more attributes that, taken collectively, allows us to identify uniquely an entity in the entity set. We choose a minimal superkey for each entity set from among its superkeys; the minimal superkey is termed the entity set's **primary key**. Similarly, a relationship set is a set of one or more attributes that, taken collectively, allows us to identify uniquely a relationship in the relationship set. Likewise, we choose a mini-

mal superkey for each relationship set from among its superkeys; this is the relationship set's primary key.

- An entity set that does not have sufficient attributes to form a primary key is termed a **weak entity set**. An entity set that has a primary key is termed a **strong entity set**.
- **Specialization** and **generalization** define a containment relationship between a higher-level entity set and one or more lower-level entity sets. Specialization is the result of taking a subset of a higher-level entity set to form a lower-level entity set. Generalization is the result of taking the union of two or more disjoint (lower-level) entity sets to produce a higher-level entity set. The attributes of higher-level entity sets are inherited by lower-level entity sets.
- **Aggregation** is an abstraction in which relationship sets (along with their associated entity sets) are treated as higher-level entity sets, and can participate in relationships.
- The various features of the E-R model offer the database designer numerous choices in how to best represent the enterprise being modeled. Concepts and objects may, in certain cases, be represented by entities, relationships, or attributes. Aspects of the overall structure of the enterprise may be best described by using weak entity sets, generalization, specialization, or aggregation. Often, the designer must weigh the merits of a simple, compact model versus those of a more precise, but more complex, one.
- A database that conforms to an E-R diagram can be represented by a collection of tables. For each entity set and for each relationship set in the database, there is a unique table that is assigned the name of the corresponding entity set or relationship set. Each table has a number of columns, each of which has a unique name. Converting database representation from an E-R diagram to a table format is the basis for deriving a relational-database design from an E-R diagram.
- The **unified modeling language (UML)** provides a graphical means of modeling various components of a software system. The class diagram component of UML is based on E-R diagrams. However, there are some differences between the two that one must beware of.

Review Terms

- Entity-relationship data model
- Entity
- Entity set
- Attributes
- Domain
- Simple and composite attributes
- Single-valued and multivalued attributes
- Null value
- Derived attribute
- Relationship, and relationship set
- Role

- Recursive relationship set
- Descriptive attributes
- Binary relationship set
- Degree of relationship set
- Mapping cardinality:
  - ☐ One-to-one relationship
  - ☐ One-to-many relationship
  - ☐ Many-to-one relationship
  - ☐ Many-to-many relationship
- Participation
  - ☐ Total participation
  - ☐ Partial participation
- Superkey, candidate key, and primary key
- Weak entity sets and strong entity sets
  - ☐ Discriminator attributes
  - ☐ Identifying relationship
- Specialization and generalization
  - ☐ Superclass and subclass
  - ☐ Attribute inheritance
  - ☐ Single and multiple inheritance
  - ☐ Condition-defined and user-defined membership
  - ☐ Disjoint and overlapping generalization
- Completeness constraint
  - ☐ Total and partial generalization
- Aggregation
- E-R diagram
- Unified Modeling Language (UML)

Exercises

2.1 Explain the distinctions among the terms primary key, candidate key, and superkey.

2.2 Construct an E-R diagram for a car-insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents.

2.3 Construct an E-R diagram for a hospital with a set of patients and a set of medical doctors. Associate with each patient a log of the various tests and examinations conducted.

2.4 A university registrar's office maintains data about the following entities: (a) courses, including number, title, credits, syllabus, and prerequisites; (b) course offerings, including course number, year, semester, section number, instructor(s), timings, and classroom; (c) students, including student-id, name, and program; and (d) instructors, including identification number, name, department, and title. Further, the enrollment of students in courses and grades awarded to students in each course they are enrolled for must be appropriately modeled.

Construct an E-R diagram for the registrar's office. Document all assumptions that you make about the mapping constraints.

2.5 Consider a database used to record the marks that students get in different exams of different course offerings.

a. Construct an E-R diagram that models exams as entities, and uses a ternary relationship, for the above database.

b. Construct an alternative E-R diagram that uses only a binary relationship between *students* and *course-offerings*. Make sure that only one relationship exists between a particular student and course-offering pair, yet you can represent the marks that a student gets in different exams of a course offering.

2.6 Construct appropriate tables for each of the E-R diagrams in Exercises 2.2 to 2.4.

2.7 Design an E-R diagram for keeping track of the exploits of your favourite sports team. You should store the matches played, the scores in each match, the players in each match and individual player statistics for each match. Summary statistics should be modeled as derived attributes

2.8 Extend the E-R diagram of the previous question to track the same information for all teams in a league.

2.9 Explain the difference between a weak and a strong entity set.

2.10 We can convert any weak entity set to a strong entity set by simply adding appropriate attributes. Why, then, do we have weak entity sets?

2.11 Define the concept of aggregation. Give two examples of where this concept is useful.

2.12 Consider the E-R diagram in Figure 2.29, which models an online bookstore.

a. List the entity sets and their primary keys.
b. Suppose the bookstore adds music cassettes and compact disks to its collection. The same music item may be present in cassette or compact disk format, with differing prices. Extend the E-R diagram to model this addition, ignoring the effect on shopping baskets.
c. Now extend the E-R diagram, using generalization, to model the case where a shopping basket may contain any combination of books, music cassettes, or compact disks.

2.13 Consider an E-R diagram in which the same entity set appears several times. Why is allowing this redundancy a bad practice that one should avoid whenever possible?

2.14 Consider a university database for the scheduling of classrooms for final exams. This database could be modeled as the single entity set *exam*, with attributes *course-name*, *section-number*, *room-number*, and *time*. Alternatively, one or more additional entity sets could be defined, along with relationship sets to replace some of the attributes of the *exam* entity set, as

- *course* with attributes *name*, *department*, and *c-number*
- *section* with attributes *s-number* and *enrollment*, and dependent as a weak entity set on *course*
- *room* with attributes *r-number*, *capacity*, and *building*

a. Show an E-R diagram illustrating the use of all three additional entity sets listed.

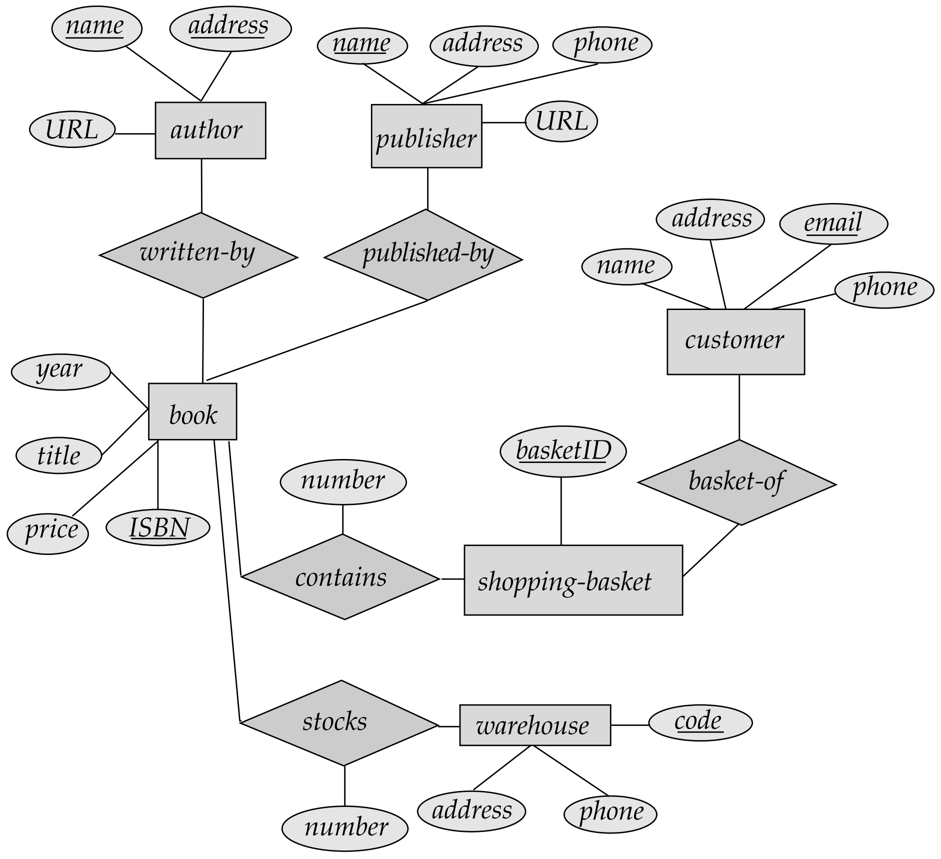


Figure 2.29 E-R diagram for Exercise 2.12.

b. Explain what application characteristics would influence a decision to include or not to include each of the additional entity sets.

2.15 When designing an E-R diagram for a particular enterprise, you have several alternatives from which to choose.

a. What criteria should you consider in making the appropriate choice?
b. Design three alternative E-R diagrams to represent the university registrar's office of Exercise 2.4. List the merits of each. Argue in favor of one of the alternatives.

2.16 An E-R diagram can be viewed as a graph. What do the following mean in terms of the structure of an enterprise schema?

a. The graph is disconnected.
b. The graph is acyclic.

2.17 In Section 2.4.3, we represented a ternary relationship (Figure 2.30a) using binary relationships, as shown in Figure 2.30b. Consider the alternative shown in

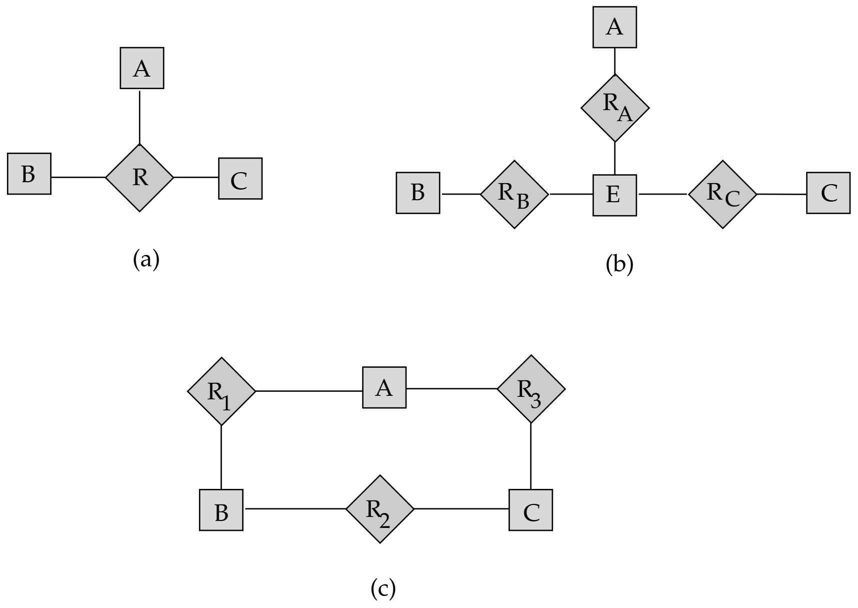


Figure 2.30 E-R diagram for Exercise 2.17 (attributes not shown).

Figure 2.30c. Discuss the relative merits of these two alternative representations of a ternary relationship by binary relationships.

2.18 Consider the representation of a ternary relationship using binary relationships as described in Section 2.4.3 (shown in Figure 2.30b.)

a. Show a simple instance of E, A, B, C, R_A, R_B, and R_C that cannot correspond to any instance of A, B, C, and R.

b. Modify the E-R diagram of Figure 2.30b to introduce constraints that will guarantee that any instance of E, A, B, C, R_A, R_B, and R_C that satisfies the constraints will correspond to an instance of A, B, C, and R.

c. Modify the translation above to handle total participation constraints on the ternary relationship.

d. The above representation requires that we create a primary key attribute for E. Show how to treat E as a weak entity set so that a primary key attribute is not required.

2.19 A weak entity set can always be made into a strong entity set by adding to its attributes the primary key attributes of its identifying entity set. Outline what sort of redundancy will result if we do so.

2.20 Design a generalization−specialization hierarchy for a motor-vehicle sales company. The company sells motorcycles, passenger cars, vans, and buses. Justify your placement of attributes at each level of the hierarchy. Explain why they should not be placed at a higher or lower level.

2.21 Explain the distinction between condition-defined and user-defined constraints. Which of these constraints can the system check automatically? Explain your answer.

2.22 Explain the distinction between disjoint and overlapping constraints.

2.23 Explain the distinction between total and partial constraints.

2.24 Figure 2.31 shows a lattice structure of generalization and specialization. For entity sets A, B, and C, explain how attributes are inherited from the higher-level entity sets X and Y. Discuss how to handle a case where an attribute of X has the same name as some attribute of Y.

2.25 Draw the UML equivalents of the E-R diagrams of Figures 2.9c, 2.10, 2.12, 2.13 and 2.17.

2.26 Consider two separate banks that decide to merge. Assume that both banks use exactly the same E-R database schema—the one in Figure 2.22. (This assumption is, of course, highly unrealistic; we consider the more realistic case in Section 19.8.) If the merged bank is to have a single database, there are several potential problems:

- The possibility that the two original banks have branches with the same name
- The possibility that some customers are customers of both original banks
- The possibility that some loan or account numbers were used at both original banks (for different loans or accounts, of course)

For each of these potential problems, describe why there is indeed a potential for difficulties. Propose a solution to the problem. For your solution, explain any changes that would have to be made and describe what their effect would be on the schema and the data.

2.27 Reconsider the situation described for Exercise 2.26 under the assumption that one bank is in the United States and the other is in Canada. As before, the banks use the schema of Figure 2.22, except that the Canadian bank uses the *social-insurance* number assigned by the Canadian government, whereas the U.S. bank uses the social-security number to identify customers. What problems (be-

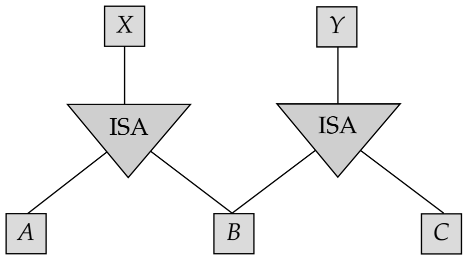


Figure 2.31 E-R diagram for Exercise 2.24 (attributes not shown).

yond those identified in Exercise 2.24) might occur in this multinational case? How would you resolve them? Be sure to consider both the scheme and the actual data values in constructing your answer.

Bibliographical Notes

The E-R data model was introduced by Chen [1976]. A logical design methodology for relational databases using the extended E-R model is presented by Teorey et al. [1986]. Mapping from extended E-R models to the relational model is discussed by Lyngbaek and Vianu [1987] and Markowitz and Shoshani [1992]. Various data-manipulation languages for the E-R model have been proposed: GERM (Benneworth et al. [1981]), GORDAS (Elmasri and Wiederhold [1981]), and ERROL (Markowitz and Raz [1983]). A graphical query language for the E-R database was proposed by Zhang and Mendelzon [1983] and Elmasri and Larson [1985].

Smith and Smith [1977] introduced the concepts of generalization, specialization, and aggregation and Hammer and McLeod [1980] expanded them. Lenzerini and Santucci [1983] used the concepts in defining cardinality constraints in the E-R model.

Thalheim [2000] provides a detailed textbook coverage of research in E-R modeling. Basic textbook discussions are offered by Batini et al. [1992] and Elmasri and Navathe [2000]. Davis et al. [1983] provide a collection of papers on the E-R model.

Tools

Many database systems provide tools for database design that support E-R diagrams. These tools help a designer create E-R diagrams, and they can automatically create corresponding tables in a database. See bibliographic notes of Chapter 1 for references to database system vendor's Web sites. There are also some database-independent data modeling tools that support E-R diagrams and UML class diagrams. These include Rational Rose (www.rational.com/products/rose), Visio Enterprise (see www.visio.com), and ERwin (search for ERwin at the site www.cai.com/products).

CHAPTER 3

Relational Model

The relational model is today the primary data model for commercial data-processing applications. It has attained its primary position because of its simplicity, which eases the job of the programmer, as compared to earlier data models such as the network model or the hierarchical model.

In this chapter, we first study the fundamentals of the relational model, which provides a very simple yet powerful way of representing data. We then describe three formal query languages; query languages are used to specify requests for information. The three we cover in this chapter are not user-friendly, but instead serve as the formal basis for user-friendly query languages that we study later. We cover the first query language, relational algebra, in great detail. The relational algebra forms the basis of the widely used SQL query language. We then provide overviews of the other two formal languages, the tuple relational calculus and the domain relational calculus, which are declarative query languages based on mathematical logic. The domain relational calculus is the basis of the QBE query language.

A substantial theory exists for relational databases. We study the part of this theory dealing with queries in this chapter. In Chapter 7 we shall examine aspects of relational database theory that help in the design of relational database schemas, while in Chapters 13 and 14 we discuss aspects of the theory dealing with efficient processing of queries.

3.1 Structure of Relational Databases

A relational database consists of a collection of **tables**, each of which is assigned a unique name. Each table has a structure similar to that presented in Chapter 2, where we represented E-R databases by tables. A row in a table represents a *relationship* among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of *table* and the mathematical concept of *relation*, from which the relational data model takes its name. In what follows, we introduce the concept of relation.

In this chapter, we shall be using a number of different relations to illustrate the various concepts underlying the relational data model. These relations represent part of a banking enterprise. They differ slightly from the tables that were used in Chapter 2, so that we can simplify our presentation. We shall discuss criteria for the appropriateness of relational structures in great detail in Chapter 7.

3.1.1 Basic Structure

Consider the *account* table of Figure 3.1. It has three column headers: *account-number*, *branch-name*, and *balance*. Following the terminology of the relational model, we refer to these headers as **attributes** (as we did for the E-R model in Chapter 2). For each attribute, there is a set of permitted values, called the **domain** of that attribute. For the attribute *branch-name*, for example, the domain is the set of all branch names. Let D_1 denote the set of all account numbers, D_2 the set of all branch names, and D_3 the set of all balances. As we saw in Chapter 2, any row of *account* must consist of a 3-tuple $(v_1,\ v_2,\ v_3)$, where v_1 is an account number (that is, v_1 is in domain D_1), v_2 is a branch name (that is, v_2 is in domain D_2), and v_3 is a balance (that is, v_3 is in domain D_3). In general, *account* will contain only a subset of the set of all possible rows. Therefore, *account* is a subset of

$$D_1 \times D_2 \times D_3$$

In general, a **table** of *n* attributes must be a subset of

$$D_1 \times D_2 \times \cdots \times D_{n-1} \times D_n$$

Mathematicians define a **relation** to be a subset of a Cartesian product of a list of domains. This definition corresponds almost exactly with our definition of *table*. The only difference is that we have assigned names to attributes, whereas mathematicians rely on numeric "names," using the integer 1 to denote the attribute whose domain appears first in the list of domains, 2 for the attribute whose domain appears second, and so on. Because tables are essentially relations, we shall use the mathematical

account-number	*branch-name*	*balance*
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

Figure 3.1 The *account* relation.

account-number	*branch-name*	*balance*
A-101	Downtown	500
A-215	Mianus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Brighton	900
A-222	Redwood	700
A-217	Brighton	750

Figure 3.2 The *account* relation with unordered tuples.

terms **relation** and **tuple** in place of the terms **table** and **row**. A **tuple variable** is a variable that stands for a tuple; in other words, a tuple variable is a variable whose domain is the set of all tuples.

In the *account* relation of Figure 3.1, there are seven tuples. Let the tuple variable t refer to the first tuple of the relation. We use the notation t[*account-number*] to denote the value of t on the *account-number* attribute. Thus, t[*account-number*] = "A-101," and t[*branch-name*] = "Downtown". Alternatively, we may write $t[1]$ to denote the value of tuple t on the first attribute (*account-number*), $t[2]$ to denote *branch-name*, and so on. Since a relation is a set of tuples, we use the mathematical notation of $t \in r$ to denote that tuple t is in relation r.

The order in which tuples appear in a relation is irrelevant, since a relation is a *set* of tuples. Thus, whether the tuples of a relation are listed in sorted order, as in Figure 3.1, or are unsorted, as in Figure 3.2, does not matter; the relations in the two figures above are the same, since both contain the same set of tuples.

We require that, for all relations r, the domains of all attributes of r be atomic. A domain is **atomic** if elements of the domain are considered to be indivisible units. For example, the set of integers is an atomic domain, but the set of all sets of integers is a nonatomic domain. The distinction is that we do not normally consider integers to have subparts, but we consider sets of integers to have subparts—namely, the integers composing the set. The important issue is not what the domain itself is, but rather how we use domain elements in our database. The domain of all integers would be nonatomic if we considered each integer to be an ordered list of digits. In all our examples, we shall assume atomic domains. In Chapter 9, we shall discuss extensions to the relational data model to permit nonatomic domains.

It is possible for several attributes to have the same domain. For example, suppose that we have a relation *customer* that has the three attributes *customer-name*, *customer-street*, and *customer-city*, and a relation *employee* that includes the attribute *employee-name*. It is possible that the attributes *customer-name* and *employee-name* will have the same domain: the set of all person names, which at the physical level is the set of all character strings. The domains of *balance* and *branch-name*, on the other hand, certainly ought to be distinct. It is perhaps less clear whether *customer-name* and *branch-name* should have the same domain. At the physical level, both customer names and branch names are character strings. However, at the logical level, we may want *customer-name* and *branch-name* to have distinct domains.

One domain value that is a member of any possible domain is the **null** value, which signifies that the value is unknown or does not exist. For example, suppose that we include the attribute *telephone-number* in the *customer* relation. It may be that a customer does not have a telephone number, or that the telephone number is unlisted. We would then have to resort to null values to signify that the value is unknown or does not exist. We shall see later that null values cause a number of difficulties when we access or update the database, and thus should be eliminated if at all possible. We shall assume null values are absent initially, and in Section 3.3.4, we describe the effect of nulls on different operations.

3.1.2 Database Schema

When we talk about a database, we must differentiate between the **database schema**, which is the logical design of the database, and a **database instance**, which is a snapshot of the data in the database at a given instant in time.

The concept of a relation corresponds to the programming-language notion of a variable. The concept of a **relation schema** corresponds to the programming-language notion of type definition.

It is convenient to give a name to a relation schema, just as we give names to type definitions in programming languages. We adopt the convention of using lowercase names for relations, and names beginning with an uppercase letter for relation schemas. Following this notation, we use *Account-schema* to denote the relation schema for relation *account*. Thus,

$$Account\text{-}schema = (\textit{account-number}, \textit{branch-name}, \textit{balance})$$

We denote the fact that *account* is a relation on *Account-schema* by

$$account(Account\text{-}schema)$$

In general, a relation schema consists of a list of attributes and their corresponding domains. We shall not be concerned about the precise definition of the domain of each attribute until we discuss the SQL language in Chapter 4.

The concept of a **relation instance** corresponds to the programming language notion of a value of a variable. The value of a given variable may change with time; similarly the contents of a relation instance may change with time as the relation is updated. However, we often simply say "relation" when we actually mean "relation instance."

As an example of a relation instance, consider the *branch* relation of Figure 3.3. The schema for that relation is

$$Branch\text{-}schema = (\textit{branch-name}, \textit{branch-city}, \textit{assets})$$

Note that the attribute *branch-name* appears in both *Branch-schema* and *Account-schema*. This duplication is not a coincidence. Rather, using common attributes in relation schemas is one way of relating tuples of distinct relations. For example, suppose we wish to find the information about all of the accounts maintained in branches

branch-name	*branch-city*	*assets*
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

Figure 3.3 The *branch* relation.

located in Brooklyn. We look first at the *branch* relation to find the names of all the branches located in Brooklyn. Then, for each such branch, we would look in the *account* relation to find the information about the accounts maintained at that branch. This is not surprising—recall that the primary key attributes of a strong entity set appear in the table created to represent the entity set, as well as in the tables created to represent relationships that the entity set participates in.

Let us continue our banking example. We need a relation to describe information about customers. The relation schema is

$$Customer\text{-}schema = (\textit{customer-name}, \textit{customer-street}, \textit{customer-city})$$

Figure 3.4 shows a sample relation *customer* (*Customer-schema*). Note that we have omitted the *customer-id* attribute, which we used Chapter 2, because now we want to have smaller relation schemas in our running example of a bank database. We assume that the customer name uniquely identifies a customer—obviously this may not be true in the real world, but the assumption makes our examples much easier to read.

customer-name	*customer-street*	*customer-city*
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

Figure 3.4 The *customer* relation.

In a real-world database, the *customer-id* (which could be a *social-security* number, or an identifier generated by the bank) would serve to uniquely identify customers.

We also need a relation to describe the association between customers and accounts. The relation schema to describe this association is

$$Depositor\text{-}schema = (\textit{customer-name}, \textit{account-number})$$

Figure 3.5 shows a sample relation *depositor* (*Depositor-schema*).

It would appear that, for our banking example, we could have just one relation schema, rather than several. That is, it may be easier for a user to think in terms of one relation schema, rather than in terms of several. Suppose that we used only one relation for our example, with schema

(*branch-name*, *branch-city*, *assets*, *customer-name*, *customer-street*
customer-city, *account-number*, *balance*)

Observe that, if a customer has several accounts, we must list her address once for each account. That is, we must repeat certain information several times. This repetition is wasteful and is avoided by the use of several relations, as in our example.

In addition, if a branch has no accounts (a newly created branch, say, that has no customers yet), we cannot construct a complete tuple on the preceding single relation, because no data concerning *customer* and *account* are available yet. To represent incomplete tuples, we must use *null* values that signify that the value is unknown or does not exist. Thus, in our example, the values for *customer-name*, *customer-street*, and so on must be null. By using several relations, we can represent the branch information for a bank with no customers without using null values. We simply use a tuple on *Branch-schema* to represent the information about the branch, and create tuples on the other schemas only when the appropriate information becomes available.

In Chapter 7, we shall study criteria to help us decide when one set of relation schemas is more appropriate than another, in terms of information repetition and the existence of null values. For now, we shall assume that the relation schemas are given.

We include two additional relations to describe data about loans maintained in the various branches in the bank:

customer-name	*account-number*
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Figure 3.5 The *depositor* relation.

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

Figure 3.6 The *loan* relation.

$$Loan\text{-}schema = (\textit{loan-number}, \textit{branch-name}, \textit{amount})$$
$$Borrower\text{-}schema = (\textit{customer-name}, \textit{loan-number})$$

Figures 3.6 and 3.7, respectively, show the sample relations *loan* (*Loan-schema*) and *borrower* (*Borrower-schema*).

The E-R diagram in Figure 3.8 depicts the banking enterprise that we have just described. The relation schemas correspond to the set of tables that we might generate by the method outlined in Section 2.9. Note that the tables for *account-branch* and *loan-branch* have been combined into the tables for *account* and *loan* respectively. Such combining is possible since the relationships are many to one from *account* and *loan*, respectively, to *branch*, and, further, the participation of *account* and *loan* in the corresponding relationships is total, as the double lines in the figure indicate. Finally, we note that the *customer* relation may contain information about customers who have neither an account nor a loan at the bank.

The banking enterprise described here will serve as our primary example in this chapter and in subsequent ones. On occasion, we shall need to introduce additional relation schemas to illustrate particular points.

3.1.3 Keys

The notions of *superkey*, *candidate key*, and *primary key*, as discussed in Chapter 2, are also applicable to the relational model. For example, in *Branch-schema*, {*branch-*

customer-name	*loan-number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figure 3.7 The *borrower* relation.

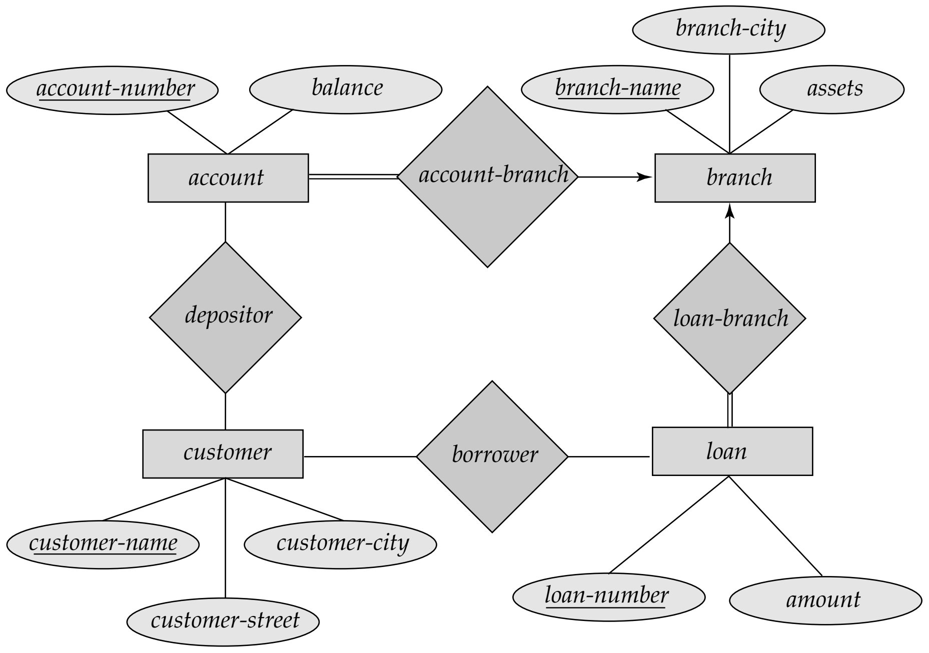


Figure 3.8 E-R diagram for the banking enterprise.

name} and {*branch-name, branch-city*} are both superkeys. {*branch-name*, *branch-city*} is not a candidate key, because {*branch-name*} is a subset of {*branch-name, branch-city*} and {*branch-name*} itself is a superkey. However, {*branch-name*} *is* a candidate key, and for our purpose also will serve as a primary key. The attribute *branch-city* is not a superkey, since two branches in the same city may have different names (and different asset figures).

Let *R* be a relation schema. If we say that a subset *K* of *R* is a *superkey* for *R*, we are restricting consideration to relations *r*(*R*) in which no two distinct tuples have the same values on all attributes in *K*. That is, if t_1 *and* t_2 are in *r* and $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$.

If a relational database schema is based on tables derived from an E-R schema, it is possible to determine the primary key for a relation schema from the primary keys of the entity or relationship sets from which the schema is derived:

- **Strong entity set**. The primary key of the entity set becomes the primary key of the relation.
- **Weak entity set**. The table, and thus the relation, corresponding to a weak entity set includes
  - ☐ The attributes of the weak entity set
  - ☐ The primary key of the strong entity set on which the weak entity set depends

The primary key of the relation consists of the union of the primary key of the strong entity set and the discriminator of the weak entity set.

- **Relationship set**. The union of the primary keys of the related entity sets becomes a superkey of the relation. If the relationship is many-to-many, this superkey is also the primary key. Section 2.4.2 describes how to determine the primary keys in other cases. Recall from Section 2.9.3 that no table is generated for relationship sets linking a weak entity set to the corresponding strong entity set.

- **Combined tables**. Recall from Section 2.9.3 that a binary many-to-one relationship set from *A* to *B* can be represented by a table consisting of the attributes of *A* and attributes (if any exist) of the relationship set. The primary key of the "many" entity set becomes the primary key of the relation (that is, if the relationship set is many to one from *A* to *B*, the primary key of *A* is the primary key of the relation). For one-to-one relationship sets, the relation is constructed like that for a many-to-one relationship set. However, we can choose either entity set's primary key as the primary key of the relation, since both are candidate keys.

- **Multivalued attributes**. Recall from Section 2.9.5 that a multivalued attribute *M* is represented by a table consisting of the primary key of the entity set or relationship set of which *M* is an attribute plus a column *C* holding an individual value of *M*. The primary key of the entity or relationship set, together with the attribute *C*, becomes the primary key for the relation.

From the preceding list, we see that a relation schema, say r_1, derived from an E-R schema may include among its attributes the primary key of another relation schema, say r_2. This attribute is called a **foreign key** from r_1, referencing r_2. The relation r_1 is also called the **referencing relation** of the foreign key dependency, and r_2 is called the **referenced relation** of the foreign key. For example, the attribute *branch-name* in *Account-schema* is a foreign key from *Account-schema* referencing *Branch-schema*, since *branch-name* is the primary key of *Branch-schema*. In any database instance, given any tuple, say t_a, from the *account* relation, there must be some tuple, say t_b, in the *branch* relation such that the value of the *branch-name* attribute of t_a is the same as the value of the primary key, *branch-name*, of t_b.

It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *branch-name* attribute of *Branch-schema* is listed first, since it is the primary key.

3.1.4 Schema Diagram

A database schema, along with primary key and foreign key dependencies, can be depicted pictorially by **schema diagram**s. Figure 3.9 shows the schema diagram for our banking enterprise. Each relation appears as a box, with the attributes listed inside it and the relation name above it. If there are primary key attributes, a horizontal line crosses the box, with the primary key attributes listed above the line. Foreign

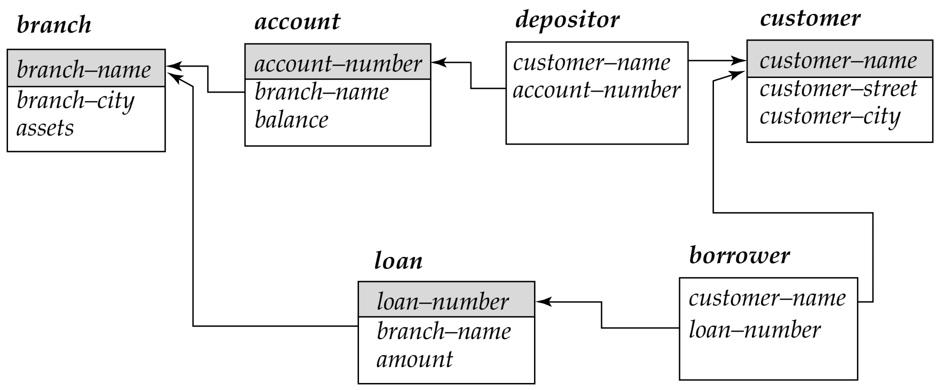


Figure 3.9 Schema diagram for the banking enterprise.

key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

Do not confuse a schema diagram with an E-R diagram. In particular, E-R diagrams do not show foreign key attributes explicitly, whereas schema diagrams show them explicity.

Many database systems provide design tools with a graphical user interface for creating schema diagrams.

3.1.5 Query Languages

A **query language** is a language in which a user requests information from the database. These languages are usually on a level higher than that of a standard programming language. Query languages can be categorized as either procedural or nonprocedural. In a **procedural language**, the user instructs the system to perform a sequence of operations on the database to compute the desired result. In a **nonprocedural language**, the user describes the desired information without giving a specific procedure for obtaining that information.

Most commercial relational-database systems offer a query language that includes elements of both the procedural and the nonprocedural approaches. We shall study the very widely used query language SQL in Chapter 4. Chapter 5 covers the query languages QBE and Datalog, the latter a query language that resembles the Prolog programming language.

In this chapter, we examine "pure" languages: The relational algebra is procedural, whereas the tuple relational calculus and domain relational calculus are nonprocedural. These query languages are terse and formal, lacking the "syntactic sugar" of commercial languages, but they illustrate the fundamental techniques for extracting data from the database.

Although we shall be concerned with only queries initially, a complete data-manipulation language includes not only a query language, but also a language for database modification. Such languages include commands to insert and delete tuples, as well as commands to modify parts of existing tuples. We shall examine database modification after we complete our discussion of queries.

3.2 The Relational Algebra

The relational algebra is a *procedural* query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result. The fundamental operations in the relational algebra are *select*, *project*, *union*, *set difference*, *Cartesian product*, and *rename*. In addition to the fundamental operations, there are several other operations—namely, set intersection, natural join, division, and assignment. We will define these operations in terms of the fundamental operations.

3.2.1 Fundamental Operations

The select, project, and rename operations are called *unary* operations, because they operate on one relation. The other three operations operate on pairs of relations and are, therefore, called *binary* operations.

3.2.1.1 The Select Operation

The **select** operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ. The argument relation is in parentheses after the σ. Thus, to select those tuples of the *loan* relation where the branch is “Perryridge,” we write

$$\sigma_{branch\text{-}name = \text{“Perryridge”}} (loan)$$

If the *loan* relation is as shown in Figure 3.6, then the relation that results from the preceding query is as shown in Figure 3.10.

We can find all tuples in which the amount lent is more than \$1200 by writing

$$\sigma_{amount > 1200} (loan)$$

In general, we allow comparisons using $=, \neq, <, \leq, >, \geq$ in the selection predicate. Furthermore, we can combine several predicates into a larger predicate by using the connectives *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). Thus, to find those tuples pertaining to loans of more than \$1200 made by the Perryridge branch, we write

$$\sigma_{branch\text{-}name = \text{“Perryridge”} \wedge amount > 1200} (loan)$$

loan-number	*branch-name*	*amount*
L-15	Perryridge	1500
L-16	Perryridge	1300

Figure 3.10 Result of $\sigma_{branch\text{-}name = \text{“Perryridge”}} (loan)$.

The selection predicate may include comparisons between two attributes. To illustrate, consider the relation *loan-officer* that consists of three attributes: *customer-name*, *banker-name*, and *loan-number*, which specifies that a particular banker is the loan officer for a loan that belongs to some customer. To find all customers who have the same name as their loan officer, we can write

$$\sigma_{customer\text{-}name\,=\,banker\text{-}name}(loan\text{-}officer)$$

3.2.1.2 The Project Operation

Suppose we want to list all loan numbers and the amount of the loans, but do not care about the branch name. The **project** operation allows us to produce this relation. The project operation is a unary operation that returns its argument relation, with certain attributes left out. Since a relation is a set, any duplicate rows are eliminated. Projection is denoted by the uppercase Greek letter pi (Π). We list those attributes that we wish to appear in the result as a subscript to Π. The argument relation follows in parentheses. Thus, we write the query to list all loan numbers and the amount of the loan as

$$\Pi_{loan\text{-}number,\ amount}(loan)$$

Figure 3.11 shows the relation that results from this query.

3.2.1.3 Composition of Relational Operations

The fact that the result of a relational operation is itself a relation is important. Consider the more complicated query "Find those customers who live in Harrison." We write:

$$\Pi_{customer\text{-}name}\ (\sigma_{customer\text{-}city\,=\,\text{“Harrison”}}\ (customer))$$

Notice that, instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.

In general, since the result of a relational-algebra operation is of the same type (relation) as its inputs, relational-algebra operations can be composed together into

loan-number	*amount*
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

Figure 3.11 Loan number and the amount of the loan.

a **relational-algebra expression**. Composing relational-algebra operations into relational-algebra expressions is just like composing arithmetic operations (such as $+$, $-$, $*$, and $\div$) into arithmetic expressions. We study the formal definition of relational-algebra expressions in Section 3.2.2.

3.2.1.4 The Union Operation

Consider a query to find the names of all bank customers who have either an account or a loan or both. Note that the *customer* relation does not contain the information, since a customer does not need to have either an account or a loan at the bank. To answer this query, we need the information in the *depositor* relation (Figure 3.5) and in the *borrower* relation (Figure 3.7). We know how to find the names of all customers with a loan in the bank:

$$\Pi_{customer\text{-}name}\ (borrower)$$

We also know how to find the names of all customers with an account in the bank:

$$\Pi_{customer\text{-}name}\ (depositor)$$

To answer the query, we need the **union** of these two sets; that is, we need all customer names that appear in either or both of the two relations. We find these data by the binary operation union, denoted, as in set theory, by $\cup$. So the expression needed is

$$\Pi_{customer\text{-}name}\ (borrower)\ \cup\ \Pi_{customer\text{-}name}\ (depositor)$$

The result relation for this query appears in Figure 3.12. Notice that there are 10 tuples in the result, even though there are seven distinct borrowers and six depositors. This apparent discrepancy occurs because Smith, Jones, and Hayes are borrowers as well as depositors. Since relations are sets, duplicate values are eliminated.

customer-name
Adams
Curry
Hayes
Jackson
Jones
Smith
Williams
Lindsay
Johnson
Turner

Figure 3.12 Names of all customers who have either a loan or an account.

Observe that, in our example, we took the union of two sets, both of which consisted of *customer-name* values. In general, we must ensure that unions are taken between *compatible* relations. For example, it would not make sense to take the union of the *loan* relation and the *borrower* relation. The former is a relation of three attributes; the latter is a relation of two. Furthermore, consider a union of a set of customer names and a set of cities. Such a union would not make sense in most situations. Therefore, for a union operation $r \cup s$ to be valid, we require that two conditions hold:

1. The relations *r* and *s* must be of the same arity. That is, they must have the same number of attributes.
2. The domains of the *i*th attribute of *r* and the *i*th attribute of *s* must be the same, for all i.

Note that *r* and *s* can be, in general, temporary relations that are the result of relational-algebra expressions.

3.2.1.5 The Set Difference Operation

The **set-difference** operation, denoted by $-$, allows us to find tuples that are in one relation but are not in another. The expression $r - s$ produces a relation containing those tuples in *r* but not in *s*.

We can find all customers of the bank who have an account but not a loan by writing

$$\Pi_{customer\text{-}name}\ (depositor) - \Pi_{customer\text{-}name}\ (borrower)$$

The result relation for this query appears in Figure 3.13.

As with the union operation, we must ensure that set differences are taken between *compatible* relations. Therefore, for a set difference operation $r - s$ to be valid, we require that the relations *r* and *s* be of the same arity, and that the domains of the *i*th attribute of *r* and the *i*th attribute of *s* be the same.

3.2.1.6 The Cartesian-Product Operation

The **Cartesian-product** operation, denoted by a cross ($\times$), allows us to combine information from any two relations. We write the Cartesian product of relations r_1 and r_2 as $r_1 \times r_2$.

customer-name
Johnson
Lindsay
Turner

Figure 3.13 Customers with an account but no loan.

Recall that a relation is by definition a subset of a Cartesian product of a set of domains. From that definition, we should already have an intuition about the definition of the Cartesian-product operation. However, since the same attribute name may appear in both r_1 and r_2, we need to devise a naming schema to distinguish between these attributes. We do so here by attaching to an attribute the name of the relation from which the attribute originally came. For example, the relation schema for $r = borrower \times$ *loan* is

(*borrower.customer-name*, *borrower.loan-number*, *loan.loan-number*,
loan.branch-name, *loan.amount*)

With this schema, we can distinguish *borrower.loan-number* from *loan.loan-number*. For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity. We can then write the relation schema for *r* as

(*customer-name*, *borrower.loan-number*, *loan.loan-number*,
branch-name, *amount*)

This naming convention *requires* that the relations that are the arguments of the Cartesian-product operation have distinct names. This requirement causes problems in some cases, such as when the Cartesian product of a relation with itself is desired. A similar problem arises if we use the result of a relational-algebra expression in a Cartesian product, since we shall need a name for the relation so that we can refer to the relation's attributes. In Section 3.2.1.7, we see how to avoid these problems by using a rename operation.

Now that we know the relation schema for *r* = *borrower* × *loan*, what tuples appear in *r*? As you may suspect, we construct a tuple of *r* out of each possible pair of tuples: one from the *borrower* relation and one from the *loan* relation. Thus, *r* is a large relation, as you can see from Figure 3.14, which includes only a portion of the tuples that make up *r*.

Assume that we have n_1 tuples in *borrower* and n_2 tuples in *loan*. Then, there are $n_1 * n_2$ ways of choosing a pair of tuples—one tuple from each relation; so there are $n_1 * n_2$ tuples in *r*. In particular, note that for some tuples *t* in *r*, it may be that *t*[*borrower.loan-number*] ≠ *t*[*loan.loan-number*].

In general, if we have relations $r_1(R_1)$ and $r_2(R_2)$, then $r_1 \times r_2$ is a relation whose schema is the concatenation of R_1 and R_2. Relation *R* contains all tuples *t* for which there is a tuple t_1 in r_1 and a tuple t_2 in r_2 for which $t[R_1] = t_1[R_1]$ and $t[R_2] = t_2[R_2]$.

Suppose that we want to find the names of all customers who have a loan at the Perryridge branch. We need the information in both the *loan* relation and the *borrower* relation to do so. If we write

$$\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \times loan)$$

then the result is the relation in Figure 3.15. We have a relation that pertains to only the Perryridge branch. However, the *customer-name* column may contain customers

customer-name	*borrower. loan-number*	*loan. loan-number*	*branch-name*	*amount*
Adams	L-16	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Adams	L-16	L-17	Downtown	1000
Adams	L-16	L-23	Redwood	2000
Adams	L-16	L-93	Mianus	500
Curry	L-93	L-11	Round Hill	900
Curry	L-93	L-14	Downtown	1500
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Curry	L-93	L-17	Downtown	1000
Curry	L-93	L-23	Redwood	2000
Curry	L-93	L-93	Mianus	500
Hayes	L-15	L-11		900
Hayes	L-15	L-14		1500
Hayes	L-15	L-15		1500
Hayes	L-15	L-16		1300
Hayes	L-15	L-17		1000
Hayes	L-15	L-23		2000
Hayes	L-15	L-93		500
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...
Smith	L-23	L-11	Round Hill	900
Smith	L-23	L-14	Downtown	1500
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Smith	L-23	L-17	Downtown	1000
Smith	L-23	L-23	Redwood	2000
Smith	L-23	L-93	Mianus	500
Williams	L-17	L-11	Round Hill	900
Williams	L-17	L-14	Downtown	1500
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300
Williams	L-17	L-17	Downtown	1000
Williams	L-17	L-23	Redwood	2000
Williams	L-17	L-93	Mianus	500

Figure 3.14 Result of $borrower \times loan$.

customer-name	*borrower. loan-number*	*loan. loan-number*	*branch-name*	*amount*
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Hayes	L-15	L-15	Perryridge	1500
Hayes	L-15	L-16	Perryridge	1300
Jackson	L-14	L-15	Perryridge	1500
Jackson	L-14	L-16	Perryridge	1300
Jones	L-17	L-15	Perryridge	1500
Jones	L-17	L-16	Perryridge	1300
Smith	L-11	L-15	Perryridge	1500
Smith	L-11	L-16	Perryridge	1300
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300

Figure 3.15 Result of $\sigma_{branch\text{-}name\,=\,\text{“Perryridge”}}\ (borrower\ \times\ loan)$.

who do not have a loan at the Perryridge branch. (If you do not see why that is true, recall that the Cartesian product takes all possible pairings of one tuple from *borrower* with one tuple of *loan*.)

Since the Cartesian-product operation associates *every* tuple of *loan* with every tuple of *borrower*, we know that, if a customer has a loan in the Perryridge branch, then there is some tuple in *borrower* × *loan* that contains his name, and *borrower.loan-number* = *loan.loan-number*. So, if we write

$$\sigma_{borrower.loan\text{-}number\,=\,loan.loan\text{-}number}\\ (\sigma_{branch\text{-}name\,=\,\text{“Perryridge”}}(borrower\ \times\ loan))$$

we get only those tuples of $borrower\ \times\ loan$ that pertain to customers who have a loan at the Perryridge branch.

Finally, since we want only *customer-name*, we do a projection:

$$\Pi_{customer\text{-}name}\ (\sigma_{borrower.loan\text{-}number\,=\,loan.loan\text{-}number}\\ (\sigma_{branch\text{-}name\,=\,\text{“Perryridge”}}\ (borrower\ \times\ loan)))$$

The result of this expression, shown in Figure 3.16, is the correct answer to our query.

3.2.1.7 The Rename Operation

Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them. It is useful to be able to give them names; the **rename** operator, denoted by the lowercase Greek letter rho (ρ), lets us do

customer-name
Adams
Hayes

Figure 3.16 Result of $\Pi_{customer\text{-}name}$ $(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}$ $(\sigma_{branch\text{-}name = \text{“Perryridge”}}\ (borrower\ \times\ loan)))$.

this. Given a relational-algebra expression E, the expression

$$\rho_x\ (E)$$

returns the result of expression *E* under the name *x*.

A relation r by itself is considered a (trivial) relational-algebra expression. Thus, we can also apply the rename operation to a relation r to get the same relation under a new name.

A second form of the rename operation is as follows. Assume that a relational-algebra expression E has arity n. Then, the expression

$$\rho_{x(A_1, A_2, \ldots, A_n)}\ (E)$$

returns the result of expression *E* under the name *x*, and with the attributes renamed to $A_1, A_2, \ldots, A_n$.

To illustrate renaming a relation, we consider the query “Find the largest account balance in the bank.” Our strategy is to (1) compute first a temporary relation consisting of those balances that are *not* the largest and (2) take the set difference between the relation $\Pi_{balance}\ (account)$ and the temporary relation just computed, to obtain the result.

Step 1: To compute the temporary relation, we need to compare the values of all account balances. We do this comparison by computing the Cartesian product $account\ \times\ account$ and forming a selection to compare the value of any two balances appearing in one tuple. First, we need to devise a mechanism to distinguish between the two *balance* attributes. We shall use the rename operation to rename one reference to the account relation; thus we can reference the relation twice without ambiguity.

balance
500
400
700
750
350

Figure 3.17 Result of the subexpression $\Pi_{account.balance}\ (\sigma_{account.balance\ <\ d.balance}\ (account\ \times\ \rho_d\ (account)))$.

balance
900

Figure 3.18 Largest account balance in the bank.

We can now write the temporary relation that consists of the balances that are not the largest:

$$\Pi_{account.balance}\ (\sigma_{account.balance\ <\ d.balance}\ (account\ \times\ \rho_d\ (account)))$$

This expression gives those balances in the *account* relation for which a larger balance appears somewhere in the *account* relation (renamed as *d*). The result contains all balances *except* the largest one. Figure 3.17 shows this relation.

Step 2: The query to find the largest account balance in the bank can be written as:

$$\begin{array}{l}\Pi_{balance}\ (account)\ - \\ \quad \Pi_{account.balance}\ (\sigma_{account.balance\ <\ d.balance}\ (account\ \times\ \rho_d\ (account)))\end{array}$$

Figure 3.18 shows the result of this query.

As one more example of the rename operation, consider the query "Find the names of all customers who live on the same street and in the same city as Smith." We can obtain Smith's street and city by writing

$$\Pi_{customer\text{-}street,\ customer\text{-}city}\ (\sigma_{customer\text{-}name\ =\ \text{“Smith”}}\ (customer))$$

However, in order to find other customers with this street and city, we must reference the *customer* relation a second time. In the following query, we use the rename operation on the preceding expression to give its result the name *smith-addr*, and to rename its attributes to *street* and *city*, instead of *customer-street* and *customer-city*:

$$\begin{array}{l}\Pi_{customer.customer\text{-}name} \\ \quad (\sigma_{customer.customer\text{-}street=smith\text{-}addr.street\ \wedge\ customer.customer\text{-}city=smith\text{-}addr.city} \\ \quad\quad (customer\ \times\ \rho_{smith\text{-}addr(street,city)} \\ \quad\quad\quad (\Pi_{customer\text{-}street,\ customer\text{-}city}\ (\sigma_{customer\text{-}name\ =\ \text{“Smith”}}(customer)))))\end{array}$$

The result of this query, when we apply it to the *customer* relation of Figure 3.4, appears in Figure 3.19.

The rename operation is not strictly required, since it is possible to use a positional notation for attributes. We can name attributes of a relation implicitly by using a positional notation, where \$1, \$2, . . . refer to the first attribute, the second attribute, and so on. The positional notation also applies to results of relational-algebra operations.

customer-name
Curry
Smith

Figure 3.19 Customers who live on the same street and in the same city as Smith.

The following relational-algebra expression illustrates the use of positional notation with the unary operator σ:

$$\sigma_{\$2=\$3}(R \times R)$$

If a binary operation needs to distinguish between its two operand relations, a similar positional notation can be used for relation names as well. For example, $\$R1$ could refer to the first operand, and $\$R2$ could refer to the second operand. However, the positional notation is inconvenient for humans, since the position of the attribute is a number, rather than an easy-to-remember attribute name. Hence, we do not use the positional notation in this textbook.

3.2.2 Formal Definition of the Relational Algebra

The operations in Section 3.2.1 allow us to give a complete definition of an expression in the relational algebra. A basic expression in the relational algebra consists of either one of the following:

- A relation in the database
- A constant relation

A constant relation is written by listing its tuples within { }, for example { (A-101, Downtown, 500) (A-215, Mianus, 700) }.

A general expression in relational algebra is constructed out of smaller subexpressions. Let E_1 and E_2 be relational-algebra expressions. Then, these are all relational-algebra expressions:

- $E_1 \cup E_2$
- $E_1 - E_2$
- $E_1 \times E_2$
- $\sigma_P(E_1)$, where P is a predicate on attributes in E_1
- $\Pi_S(E_1)$, where S is a list consisting of some of the attributes in E_1
- $\rho_x\ (E_1)$, where x is the new name for the result of E_1

3.2.3 Additional Operations

The fundamental operations of the relational algebra are sufficient to express any relational-algebra query.[1] However, if we restrict ourselves to just the fundamental operations, certain common queries are lengthy to express. Therefore, we define additional operations that do not add any power to the algebra, but simplify common queries. For each new operation, we give an equivalent expression that uses only the fundamental operations.

1. In Section 3.3, we introduce operations that extend the power of the relational algebra, to handle null and aggregate values.

3.2.3.1 The Set-Intersection Operation

The first additional-relational algebra operation that we shall define is **set intersection** (∩). Suppose that we wish to find all customers who have both a loan and an account. Using set intersection, we can write

$$\Pi_{customer\text{-}name}\ (borrower)\ \cap\ \Pi_{customer\text{-}name}\ (depositor)$$

The result relation for this query appears in Figure 3.20.

Note that we can rewrite any relational algebra expression that uses set intersection by replacing the intersection operation with a pair of set-difference operations as:

$$r\ \cap\ s = r\ -\ (r\ -\ s)$$

Thus, set intersection is not a fundamental operation and does not add any power to the relational algebra. It is simply more convenient to write $r\ \cap\ s$ than to write $r\ -\ (r\ -\ s)$.

3.2.3.2 The Natural-Join Operation

It is often desirable to simplify certain queries that require a Cartesian product. Usually, a query that involves a Cartesian product includes a selection operation on the result of the Cartesian product. Consider the query "Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount." We first form the Cartesian product of the *borrower* and *loan* relations. Then, we select those tuples that pertain to only the same *loan-number*, followed by the projection of the resulting *customer-name*, *loan-number*, and *amount*:

$$\Pi_{customer\text{-}name,\ loan.loan\text{-}number,\ amount}$$
$$(\sigma_{borrower.loan\text{-}number\ =\ loan.loan\text{-}number}\ (borrower\ \times\ loan))$$

The *natural join* is a binary operation that allows us to combine certain selections and a Cartesian product into one operation. It is denoted by the "join" symbol ⋈. The natural-join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes.

Although the definition of natural join is complicated, the operation is easy to apply. As an illustration, consider again the example "Find the names of all customers who have a loan at the bank, and find the amount of the loan." We express this query

customer-name
Hayes
Jones
Smith

Figure 3.20 Customers with both an account and a loan at the bank.

customer-name	*loan-number*	*amount*
Adams	L-16	1300
Curry	L-93	500
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-17	1000
Smith	L-23	2000
Smith	L-11	900
Williams	L-17	1000

Figure 3.21 Result of $\Pi_{customer\text{-}name,\ loan\text{-}number,\ amount}\ (borrower \bowtie loan)$.

by using the natural join as follows:

$$\Pi_{customer\text{-}name,\ loan\text{-}number,\ amount}\ (borrower \bowtie loan)$$

Since the schemas for *borrower* and *loan* (that is, *Borrower-schema* and *Loan-schema*) have the attribute *loan-number* in common, the natural-join operation considers only pairs of tuples that have the same value on *loan-number*. It combines each such pair of tuples into a single tuple on the union of the two schemas (that is, *customer-name, branch-name, loan-number, amount*). After performing the projection, we obtain the relation in Figure 3.21.

Consider two relation schemas R and S—which are, of course, lists of attribute names. If we consider the schemas to be *sets*, rather than lists, we can denote those attribute names that appear in both R and S by $R \cap S$, and denote those attribute names that appear in R, in S, or in both by $R \cup S$. Similarly, those attribute names that appear in R but not S are denoted by $R - S$, whereas $S - R$ denotes those attribute names that appear in S but not in R. Note that the union, intersection, and difference operations here are on sets of attributes, rather than on relations.

We are now ready for a formal definition of the natural join. Consider two relations $r(R)$ and $s(S)$. The **natural join** of r and s, denoted by $r \bowtie s$, is a relation on schema $R \cup S$ formally defined as follows:

$$r \bowtie s = \Pi_{R \cup S}\ (\sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \ldots \wedge r.A_n = s.A_n}\ r \times s)$$

where $R \cap S = \{A_1,\ A_2, \ldots, A_n\}$.

Because the natural join is central to much of relational-database theory and practice, we give several examples of its use.

branch-name
Brighton
Perryridge

Figure 3.22 Result of $\Pi_{branch\text{-}name}(\sigma_{customer\text{-}city = \text{“Harrison”}}\ (customer \bowtie account \bowtie depositor))$.

- Find the names of all branches with customers who have an account in the bank and who live in Harrison.

  $$\Pi_{branch\text{-}name} \\ (\sigma_{customer\text{-}city\,=\,\text{“Harrison”}}\ (customer \bowtie\ account \bowtie\ depositor))$$

  The result relation for this query appears in Figure 3.22.

  Notice that we wrote *customer* ⋈ *account* ⋈ *depositor* without inserting parentheses to specify the order in which the natural-join operations on the three relations should be executed. In the preceding case, there are two possibilities:

  □ $(customer\ \bowtie\ account)\ \bowtie\ depositor$
  □ $customer\ \bowtie\ (account\ \bowtie\ depositor)$

  We did not specify which expression we intended, because the two are equivalent. That is, the natural join is **associative**.

- Find all customers who have *both* a loan and an account at the bank.

  $$\Pi_{customer\text{-}name}\ (borrower\ \bowtie\ depositor)$$

  Note that in Section 3.2.3.1 we wrote an expression for this query by using set intersection. We repeat this expression here.

  $$\Pi_{customer\text{-}name}\ (borrower)\ \cap\ \Pi_{customer\text{-}name}\ (depositor)$$

  The result relation for this query appeared earlier in Figure 3.20. This example illustrates a general fact about the relational algebra: It is possible to write several equivalent relational-algebra expressions that are quite different from one another.

- Let $r(R)$ and $s(S)$ be relations without any attributes in common; that is, $R\ \cap\ S = \emptyset$. ($\emptyset$ denotes the empty set.) Then, $r\ \bowtie\ s\ =\ r\ \times\ s$.

The *theta join* operation is an extension to the natural-join operation that allows us to combine a selection and a Cartesian product into a single operation. Consider relations $r(R)$ and $s(S)$, and let θ be a predicate on attributes in the schema $R \cup S$. The **theta join** operation $r \bowtie_\theta s$ is defined as follows:

$$r \bowtie_\theta s\ =\ \sigma_\theta(r\ \times\ s)$$

3.2.3.3 The Division Operation

The **division** operation, denoted by $\div$, is suited to queries that include the phrase “for all.” Suppose that we wish to find all customers who have an account at *all* the branches located in Brooklyn. We can obtain all branches in Brooklyn by the expression

$$r_1\ =\ \Pi_{branch\text{-}name}\ (\sigma_{branch\text{-}city\,=\,\text{“Brooklyn”}}\ (branch))$$

The result relation for this expression appears in Figure 3.23.

branch-name
Brighton
Downtown

Figure 3.23 Result of $\Pi_{branch\text{-}name}(\sigma_{branch\text{-}city\,=\,\text{“Brooklyn”}}\ (branch)$.

We can find all (*customer-name, branch-name*) pairs for which the customer has an account at a branch by writing

$$r_2 \ = \ \Pi_{customer\text{-}name,\ branch\text{-}name}\ (depositor \bowtie \ account)$$

Figure 3.24 shows the result relation for this expression.

Now, we need to find customers who appear in r_2 with *every* branch name in r_1. The operation that provides exactly those customers is the divide operation. We formulate the query by writing

$$\begin{array}{l}\Pi_{customer\text{-}name,\ branch\text{-}name}\ (depositor \bowtie \ account)\\ \quad \div\ \Pi_{branch\text{-}name}\ (\sigma_{branch\text{-}city\,=\,\text{“Brooklyn”}}\ (branch))\end{array}$$

The result of this expression is a relation that has the schema (*customer-name*) and that contains the tuple (Johnson).

Formally, let $r(R)$ and $s(S)$ be relations, and let $S \ \subseteq \ R$; that is, every attribute of schema S is also in schema R. The relation $r \ \div \ s$ is a relation on schema $R \ - \ S$ (that is, on the schema containing all attributes of schema R that are not in schema S). A tuple *t* is in $r \ \div \ s$ if and only if both of two conditions hold:

1. *t* is in $\Pi_{R-S}(r)$
2. For every tuple t_s in *s*, there is a tuple t_r in *r* satisfying both of the following:
   - **a.** $t_r[S] \ = \ t_s[S]$
   - **b.** $t_r[R \ - \ S] \ = \ t$

It may surprise you to discover that, given a division operation and the schemas of the relations, we can, in fact, define the division operation in terms of the fundamental operations. Let $r(R)$ and $s(S)$ be given, with $S \ \subseteq \ R$:

$$r \ \div \ s \ = \ \Pi_{R-S}\ (r) \ - \ \Pi_{R-S}\ ((\Pi_{R-S}\ (r) \ \times \ s) \ - \ \Pi_{R-S,S}(r))$$

customer-name	*branch-name*
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill

Figure 3.24 Result of $\Pi_{customer\text{-}name,\ branch\text{-}name}\ (depositor \bowtie \ account)$.

To see that this expression is true, we observe that $\Pi_{R-S}\ (r)$ gives us all tuples t that satisfy the first condition of the definition of division. The expression on the right side of the set difference operator

$$\Pi_{R-S}\ ((\Pi_{R-S}\ (r)\ \times\ s)\ -\ \Pi_{R-S,S}(r))$$

serves to eliminate those tuples that fail to satisfy the second condition of the definition of division. Let us see how it does so. Consider $\Pi_{R-S}\ (r)\ \times\ s$. This relation is on schema *R*, and pairs every tuple in $\Pi_{R-S}\ (r)$ with every tuple in s. The expression $\Pi_{R-S,S}(r)$ merely reorders the attributes of r.

Thus, $(\Pi_{R-S}\ (r)\ \times\ s)\ -\ \Pi_{R-S,S}(r)$ gives us those pairs of tuples from $\Pi_{R-S}\ (r)$ and s that do not appear in *r*. If a tuple t_j is in

$$\Pi_{R-S}\ ((\Pi_{R-S}\ (r)\ \times\ s)\ -\ \Pi_{R-S,S}(r))$$

then there is some tuple t_s in s that does not combine with tuple t_j to form a tuple in *r*. Thus, t_j holds a value for attributes $R\ -\ S$ that does not appear in $r\ \div\ s$. It is these values that we eliminate from $\Pi_{R-S}\ (r)$.

3.2.3.4 The Assignment Operation

It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables. The **assignment** operation, denoted by $\leftarrow$, works like assignment in a programming language. To illustrate this operation, consider the definition of division in Section 3.2.3.3. We could write $r\ \div\ s$ as

$$\begin{aligned}&temp1\ \leftarrow\ \Pi_{R-S}\ (r)\\&temp2\ \leftarrow\ \Pi_{R-S}\ ((temp1\ \times\ s)\ -\ \Pi_{R-S,S}(r))\\&result\ =\ temp1\ -\ temp2\end{aligned}$$

The evaluation of an assignment does not result in any relation being displayed to the user. Rather, the result of the expression to the right of the $\leftarrow$ is assigned to the relation variable on the left of the $\leftarrow$. This relation variable may be used in subsequent expressions.

With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query. For relational-algebra queries, assignment must always be made to a temporary relation variable. Assignments to permanent relations constitute a database modification. We discuss this issue in Section 3.4. Note that the assignment operation does not provide any additional power to the algebra. It is, however, a convenient way to express complex queries.

3.3 Extended Relational-Algebra Operations

The basic relational-algebra operations have been extended in several ways. A simple extension is to allow arithmetic operations as part of projection. An important extension is to allow *aggregate operations* such as computing the sum of the elements of a

customer-name	*limit*	*credit-balance*
Curry	2000	1750
Hayes	1500	1500
Jones	6000	700
Smith	2000	400

Figure 3.25 The *credit-info* relation.

set, or their average. Another important extension is the *outer-join* operation, which allows relational-algebra expressions to deal with null values, which model missing information.

3.3.1 Generalized Projection

The **generalized-projection** operation extends the projection operation by allowing arithmetic functions to be used in the projection list. The generalized projection operation has the form

$$\Pi_{F_1, F_2, \ldots, F_n}(E)$$

where E is any relational-algebra expression, and each of $F_1, F_2, \ldots, F_n$ is an arithmetic expression involving constants and attributes in the schema of E. As a special case, the arithmetic expression may be simply an attribute or a constant.

For example, suppose we have a relation *credit-info*, as in Figure 3.25, which lists the credit limit and expenses so far (the *credit-balance* on the account). If we want to find how much more each person can spend, we can write the following expression:

$$\Pi_{customer\text{-}name,\ limit\ -\ credit\text{-}balance}\ (\textit{credit-info})$$

The attribute resulting from the expression $limit\ -\ credit\text{-}balance$ does not have a name. We can apply the rename operation to the result of generalized projection in order to give it a name. As a notational convenience, renaming of attributes can be combined with generalized projection as illustrated below:

$$\Pi_{customer\text{-}name,\ (limit\ -\ credit\text{-}balance)\ \mathbf{as}\ credit\text{-}available}\ (\textit{credit-info})$$

The second attribute of this generalized projection has been given the name *credit-available*. Figure 3.26 shows the result of applying this expression to the relation in Figure 3.25.

3.3.2 Aggregate Functions

Aggregate functions take a collection of values and return a single value as a result. For example, the aggregate function **sum** takes a collection of values and returns the sum of the values. Thus, the function **sum** applied on the collection

$$\{1, 1, 3, 4, 4, 11\}$$

customer-name	*credit-available*
Curry	250
Jones	5300
Smith	1600
Hayes	0

Figure 3.26 The result of $\Pi_{customer\text{-}name,\ (limit\ -\ credit\text{-}balance)\ \textbf{as}\ credit\text{-}available}(credit\text{-}info)$.

returns the value 24. The aggregate function **avg** returns the average of the values. When applied to the preceding collection, it returns the value 4. The aggregate function **count** returns the number of the elements in the collection, and returns 6 on the preceding collection. Other common aggregate functions include **min** and **max**, which return the minimum and maximum values in a collection; they return 1 and 11, respectively, on the preceding collection.

The collections on which aggregate functions operate can have multiple occurrences of a value; the order in which the values appear is not relevant. Such collections are called **multisets**. Sets are a special case of multisets where there is only one copy of each element.

To illustrate the concept of aggregation, we shall use the *pt-works* relation in Figure 3.27, for part-time employees. Suppose that we want to find out the total sum of salaries of all the part-time employees in the bank. The relational-algebra expression for this query is:

$$\mathcal{G}_{\textbf{sum}(salary)}(pt\text{-}works)$$

The symbol $\mathcal{G}$ is the letter G in calligraphic font; read it as "calligraphic G." The relational-algebra operation $\mathcal{G}$ signifies that aggregation is to be applied, and its subscript specifies the aggregate operation to be applied. The result of the expression above is a relation with a single attribute, containing a single row with a numerical value corresponding to the sum of all the salaries of all employees working part-time in the bank.

employee-name	*branch-name*	*salary*
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Rao	Austin	1500
Sato	Austin	1600

Figure 3.27 The *pt-works* relation.

There are cases where we must eliminate multiple occurrences of a value before computing an aggregate function. If we do want to eliminate duplicates, we use the same function names as before, with the addition of the hyphenated string "**distinct**" appended to the end of the function name (for example, **count-distinct**). An example arises in the query "Find the number of branches appearing in the *pt-works* relation." In this case, a branch name counts only once, regardless of the number of employees working that branch. We write this query as follows:

$$\mathcal{G}_{\textbf{count-distinct}(branch\text{-}name)}(pt\text{-}works)$$

For the relation in Figure 3.27, the result of this query is a single row containing the value 3.

Suppose we want to find the total salary sum of all part-time employees at each branch of the bank separately, rather than the sum for the entire bank. To do so, we need to partition the relation *pt-works* into **groups** based on the branch, and to apply the aggregate function on each group.

The following expression using the aggregation operator $\mathcal{G}$ achieves the desired result:

$$_{branch\text{-}name}\mathcal{G}_{\textbf{sum}(salary)}(pt\text{-}works)$$

In the expression, the attribute *branch-name* in the left-hand subscript of $\mathcal{G}$ indicates that the input relation *pt-works* must be divided into groups based on the value of *branch-name*. Figure 3.28 shows the resulting groups. The expression **sum**$(salary)$ in the right-hand subscript of $\mathcal{G}$ indicates that for each group of tuples (that is, each branch), the aggregation function **sum** must be applied on the collection of values of the *salary* attribute. The output relation consists of tuples with the branch name, and the sum of the salaries for the branch, as shown in Figure 3.29.

The general form of the **aggregation operation** $\mathcal{G}$ is as follows:

$$_{G_1, G_2, \ldots, G_n}\mathcal{G}_{F_1(A_1),\ F_2(A_2), \ldots,\ F_m(A_m)}(E)$$

where E is any relational-algebra expression; $G_1, G_2, \ldots, G_n$ constitute a list of attributes on which to group; each F_i is an aggregate function; and each A_i is an at-

employee-name	*branch-name*	*salary*
Rao	Austin	1500
Sato	Austin	1600
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300

Figure 3.28 The *pt-works* relation after grouping.

branch-name	*sum of salary*
Austin	3100
Downtown	5300
Perryridge	8100

Figure 3.29 Result of $_{branch\text{-}name}\mathcal{G}_{\mathbf{sum}(salary)}(pt\text{-}works)$.

tribute name. The meaning of the operation is as follows. The tuples in the result of expression E are partitioned into groups in such a way that

1. All tuples in a group have the same values for $G_1, G_2, \ldots, G_n$.
2. Tuples in different groups have different values for $G_1, G_2, \ldots, G_n$.

Thus, the groups can be identified by the values of attributes $G_1, G_2, \ldots, G_n$. For each group $(g_1, g_2, \ldots, g_n)$, the result has a tuple $(g_1, g_2, \ldots, g_n, a_1, a_2, \ldots, a_m)$ where, for each i, a_i is the result of applying the aggregate function F_i on the multiset of values for attribute A_i in the group.

As a special case of the aggregate operation, the list of attributes $G_1, G_2, \ldots, G_n$ can be empty, in which case there is a single group containing all tuples in the relation. This corresponds to aggregation without grouping.

Going back to our earlier example, if we want to find the maximum salary for part-time employees at each branch, in addition to the sum of the salaries, we write the expression

$$_{branch\text{-}name}\mathcal{G}_{\mathbf{sum}(salary),\mathbf{max}(salary)}(pt\text{-}works)$$

As in generalized projection, the result of an aggregation operation does not have a name. We can apply a rename operation to the result in order to give it a name. As a notational convenience, attributes of an aggregation operation can be renamed as illustrated below:

$$_{branch\text{-}name}\mathcal{G}_{\mathbf{sum}(salary)\ \mathbf{as}\ sum\text{-}salary,\mathbf{max}(salary)\ \mathbf{as}\ max\text{-}salary}(pt\text{-}works)$$

Figure 3.30 shows the result of the expression.

branch-name	*sum-salary*	*max-salary*
Austin	3100	1600
Downtown	5300	2500
Perryridge	8100	5300

Figure 3.30 Result of $_{branch\text{-}name}\mathcal{G}_{\mathbf{sum}(salary)\ \mathbf{as}\ sum\text{-}salary,\mathbf{max}(salary)\ \mathbf{as}\ max\text{-}salary}(pt\text{-}works)$.

employee-name	*street*	*city*
Coyote	Toon	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Death Valley
Williams	Seaview	Seattle

employee-name	*branch-name*	*salary*
Coyote	Mesa	1500
Rabbit	Mesa	1300
Gates	Redmond	5300
Williams	Redmond	1500

Figure 3.31 The *employee* and *ft-works* relations.

3.3.3 Outer Join

The **outer-join** operation is an extension of the join operation to deal with missing information. Suppose that we have the relations with the following schemas, which contain data on full-time employees:

employee (*employee-name, street, city*)
ft-works (*employee-name, branch-name, salary*)

Consider the *employee* and *ft-works* relations in Figure 3.31. Suppose that we want to generate a single relation with all the information (street, city, branch name, and salary) about full-time employees. A possible approach would be to use the natural-join operation as follows:

$$employee \bowtie ft\text{-}works$$

The result of this expression appears in Figure 3.32. Notice that we have lost the street and city information about Smith, since the tuple describing Smith is absent from the *ft-works* relation; similarly, we have lost the branch name and salary information about Gates, since the tuple describing Gates is absent from the *employee* relation.

We can use the *outer-join* operation to avoid this loss of information. There are actually three forms of the operation: *left outer join*, denoted ⟕; *right outer join*, denoted ⟖; and *full outer join*, denoted ⟗. All three forms of outer join compute the join, and add extra tuples to the result of the join. The results of the expressions

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500

Figure 3.32 The result of $employee \bowtie ft\text{-}works$.

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	*null*	*null*

Figure 3.33 Result of $employee ⟕ ft\text{-}works$.

$employee ⟕ ft\text{-}works$,, $employee ⟖ ft\text{-}works$, and $employee ⟗ ft\text{-}works$ appear in Figures 3.33, 3.34, and 3.35, respectively.

The **left outer join** (⟕) takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes from the right relation, and adds them to the result of the natural join. In Figure 3.33, tuple (Smith, Revolver, Death Valley, $null$, $null$) is such a tuple. All information from the left relation is present in the result of the left outer join.

The **right outer join** (⟖) is symmetric with the left outer join: It pads tuples from the right relation that did not match any from the left relation with nulls and adds them to the result of the natural join. In Figure 3.34, tuple (Gates, $null$, $null$, Redmond, 5300) is such a tuple. Thus, all information from the right relation is present in the result of the right outer join.

The **full outer join**(⟗) does both of those operations, padding tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation, and adding them to the result of the join. Figure 3.35 shows the result of a full outer join.

Since outer join operations may generate results containing null values, we need to specify how the different relational-algebra operations deal with null values. Section 3.3.4 deals with this issue.

It is interesting to note that the outer join operations can be expressed by the basic relational-algebra operations. For instance, the left outer join operation, $r ⟕ s$, can be written as

$$(r \bowtie s) \cup (r - \Pi_R(r \bowtie s)) \times \{(null, \ldots, null)\}$$

where the constant relation $\{(null, \ldots, null)\}$ is on the schema $S - R$.

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Gates	*null*	*null*	Redmond	5300

Figure 3.34 Result of $employee ⟖ ft\text{-}works$.

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	*null*	*null*
Gates	*null*	*null*	Redmond	5300

Figure 3.35 Result of $employee$ ⟗ $ft\text{-}works$.

3.3.4 Null Values$**$

In this section, we define how the various relational algebra operations deal with null values and complications that arise when a null value participates in an arithmetic operation or in a comparison. As we shall see, there is often more than one possible way of dealing with null values, and as a result our definitions can sometimes be arbitrary. Operations and comparisons on null values should therefore be avoided, where possible.

Since the special value *null* indicates "value unknown or nonexistent," any arithmetic operations (such as $+, -, *, /$) involving null values must return a null result.

Similarly, any comparisons (such as $<, <=, >, >=, \neq$) involving a null value evaluate to special value **unknown**; we cannot say for sure whether the result of the comparison is true or false, so we say that the result is the new truth value *unknown*.

Comparisons involving nulls may occur inside Boolean expressions involving the and, or, and not operations. We must therefore define how the three Boolean operations deal with the truth value *unknown*.

- **and**: (*true* **and** *unknown*) = *unknown*; (*false* **and** *unknown*) = *false*; (*unknown* **and** *unknown*) = *unknown*.
- **or**: (*true* **or** *unknown*) = *true*; (*false* **or** *unknown*) = *unknown*; (*unknown* **or** *unknown*) = *unknown*.
- **not**: (**not** *unknown*) = *unknown*.

We are now in a position to outline how the different relational operations deal with null values. Our definitions follow those used in the SQL language.

- **select**: The selection operation evaluates predicate P in $\sigma_P(E)$ on each tuple t in E. If the predicate returns the value *true*, t is added to the result. Otherwise, if the predicate returns *unknown* or *false*, t is not added to the result.
- **join**: Joins can be expressed as a cross product followed by a selection. Thus, the definition of how selection handles nulls also defines how join operations handle nulls.

  In a natural join, say $r \bowtie s$, we can see from the above definition that if two tuples, $t_r \in r$ and $t_s \in s$, both have a null value in a common attribute, then the tuples do not match.

- **projection**: The projection operation treats nulls just like any other value when eliminating duplicates. Thus, if two tuples in the projection result are exactly the same, and both have nulls in the same fields, they are treated as duplicates.
  The decision is a little arbitrary since, without knowing the actual value, we do not know if the two instances of null are duplicates or not.
- **union**, **intersection**, **difference**: These operations treat nulls just as the projection operation does; they treat tuples that have the same values on all fields as duplicates even if some of the fields have null values in both tuples.
  The behavior is rather arbitrary, especially in the case of intersection and difference, since we do not know if the actual values (if any) represented by the nulls are the same.
- **generalized projection**: We outlined how nulls are handled in expressions at the beginning of Section 3.3.4. Duplicate tuples containing null values are handled as in the projection operation.
- **aggregate**: When nulls occur in grouping attributes, the aggregate operation treats them just as in projection: If two tuples are the same on all grouping attributes, the operation places them in the same group, even if some of their attribute values are null.
  When nulls occur in aggregated attributes, the operation deletes null values at the outset, before applying aggregation. If the resultant multiset is empty, the aggregate result is null.
  Note that the treatment of nulls here is different from that in ordinary arithmetic expressions; we could have defined the result of an aggregate operation as null if even one of the aggregated values is null. However, this would mean a single unknown value in a large group could make the aggregate result on the group to be null, and we would lose a lot of useful information.
- **outer join**: Outer join operations behave just like join operations, except on tuples that do not occur in the join result. Such tuples may be added to the result (depending on whether the operation is ⟕, ⟖, or ⟗), padded with nulls.

3.4 Modification of the Database

We have limited our attention until now to the extraction of information from the database. In this section, we address how to add, remove, or change information in the database.

We express database modifications by using the assignment operation. We make assignments to actual database relations by using the same notation as that described in Section 3.2.3 for assignment.

3.4.1 Deletion

We express a delete request in much the same way as a query. However, instead of displaying tuples to the user, we remove the selected tuples from the database. We can delete only whole tuples; we cannot delete values on only particular attributes. In relational algebra a deletion is expressed by

$$r \leftarrow r - E$$

where *r* is a relation and *E* is a relational-algebra query.

Here are several examples of relational-algebra delete requests:

- Delete all of Smith's account records.

$$depositor \leftarrow depositor - \sigma_{customer\text{-}name = \text{“Smith”}}(depositor)$$

- Delete all loans with amount in the range 0 to 50.

$$loan \leftarrow loan - \sigma_{amount \geq 0 \text{ and } amount \leq 50}(loan)$$

- Delete all accounts at branches located in Needham.

$$\begin{aligned} &r_1 \leftarrow \sigma_{branch\text{-}city = \text{“Needham”}}(account \bowtie branch)\\ &r_2 \leftarrow \Pi_{branch\text{-}name,\ account\text{-}number,\ balance}(r_1)\\ &account \leftarrow account - r_2 \end{aligned}$$

Note that, in the final example, we simplified our expression by using assignment to temporary relations (r_1 and r_2).

3.4.2 Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. Obviously, the attribute values for inserted tuples must be members of the attribute's domain. Similarly, tuples inserted must be of the correct arity. The relational algebra expresses an insertion by

$$r \leftarrow r \cup E$$

where *r* is a relation and *E* is a relational-algebra expression. We express the insertion of a single tuple by letting *E* be a constant relation containing one tuple.

Suppose that we wish to insert the fact that Smith has \$1200 in account A-973 at the Perryridge branch. We write

$$\begin{aligned} &account \leftarrow account \cup \{(\text{A-973, “Perryridge”, 1200})\}\\ &depositor \leftarrow depositor \cup \{(\text{“Smith”, A-973})\} \end{aligned}$$

More generally, we might want to insert tuples on the basis of the result of a query. Suppose that we want to provide as a gift for all loan customers of the Perryridge branch a new \$200 savings account. Let the loan number serve as the account number for this savings account. We write

$$\begin{aligned} &r_1 \leftarrow (\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \bowtie loan))\\ &r_2 \leftarrow \Pi_{loan\text{-}number,\ branch\text{-}name}(r_1)\\ &account \leftarrow account \cup (r_2 \times \{(200)\})\\ &depositor \leftarrow depositor \cup \Pi_{customer\text{-}name,\ loan\text{-}number}(r_1) \end{aligned}$$

Instead of specifying a tuple as we did earlier, we specify a set of tuples that is inserted into both the *account* and *depositor* relation. Each tuple in the *account* relation has an *account-number* (which is the same as the loan number), a *branch-name* (Perryridge), and the initial balance of the new account ($200). Each tuple in the *depositor* relation has as *customer-name* the name of the loan customer who is being given the new account and the same account number as the corresponding *account* tuple.

3.4.3 Updating

In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple. We can use the generalized-projection operator to do this task:

$$r \leftarrow \Pi_{F_1, F_2, \ldots, F_n}(r)$$

where each F_i is either the ith attribute of r, if the ith attribute is not updated, or, if the attribute is to be updated, F_i is an expression, involving only constants and the attributes of r, that gives the new value for the attribute.

If we want to select some tuples from r and to update only them, we can use the following expression; here, P denotes the selection condition that chooses which tuples to update:

$$r \leftarrow \Pi_{F_1, F_2, \ldots, F_n}(\sigma_P(r)) \cup (r - \sigma_P(r))$$

To illustrate the use of the update operation, suppose that interest payments are being made, and that all balances are to be increased by 5 percent. We write

$$account \leftarrow \Pi_{account\text{-}number,\ branch\text{-}name,\ balance\ *1.05}\ (account)$$

Now suppose that accounts with balances over $10,000 receive 6 percent interest, whereas all others receive 5 percent. We write

$$\begin{aligned} account \leftarrow\ & \Pi_{AN, BN,\ balance\ *1.06}\ (\sigma_{balance > 10000}\ (account)) \\ & \cup\ \Pi_{AN,\ BN\ balance\ *1.05}\ (\sigma_{balance \leq 10000}\ (account)) \end{aligned}$$

where the abbreviations AN and BN stand for *account-number* and *branch-name*, respectively.

3.5 Views

In our examples up to this point, we have operated at the logical-model level. That is, we have assumed that the relations in thecollection we are given are the actual relations stored in the database.

It is not desirable for all users to see the entire logical model. Security considerations may require that certain data be hidden from users. Consider a person who needs to know a customer's loan number and branch name, but has no need to see the loan amount. This person should see a relation described, in the relational algebra, by

$$\Pi_{customer\text{-}name,\ loan\text{-}number,\ branch\text{-}name}\ (borrower \bowtie\ loan)$$

Aside from security concerns, we may wish to create a personalized collection of relations that is better matched to a certain user's intuition than is the logical model.

An employee in the advertising department, for example, might like to see a relation consisting of the customers who have either an account or a loan at the bank, and the branches with which they do business. The relation that we would create for that employee is

$$\Pi_{branch\text{-}name,\ customer\text{-}name}\ (depositor \bowtie\ account)$$
$$\cup\ \Pi_{branch\text{-}name,\ customer\text{-}name}\ (borrower \bowtie\ loan)$$

Any relation that is not part of the logical model, but is made visible to a user as a virtual relation, is called a **view**. It is possible to support a large number of views on top of any given set of actual relations.

3.5.1 View Definition

We define a view by using the **create view** statement. To define a view, we must give the view a name, and must state the query that computes the view. The form of the **create view** statement is

create view v **as** <query expression>

where <query expression> is any legal relational-algebra query expression. The view name is represented by v.

As an example, consider the view consisting of branches and their customers. We wish this view to be called *all-customer*. We define this view as follows:

create view *all-customer* **as**

$$\Pi_{branch\text{-}name,\ customer\text{-}name}\ (depositor \bowtie\ account)$$
$$\cup\ \Pi_{branch\text{-}name,\ customer\text{-}name}\ (borrower \bowtie\ loan)$$

Once we have defined a view, we can use the view name to refer to the virtual relation that the view generates. Using the view *all-customer*, we can find all customers of the Perryridge branch by writing

$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name\,=\,\text{“Perryridge”}}\ (all\text{-}customer))$$

Recall that we wrote the same query in Section 3.2.1 without using views.

View names may appear in any place where a relation name may appear, so long as no update operations are executed on the views. We study the issue of update operations on views in Section 3.5.2.

View definition differs from the relational-algebra assignment operation. Suppose that we define relation $r1$ as follows:

$$r1 \leftarrow \Pi_{branch\text{-}name,\ customer\text{-}name}\ (depositor \bowtie\ account)$$
$$\cup\ \Pi_{branch\text{-}name,\ customer\text{-}name}(borrower \bowtie\ loan)$$

We evaluate the assignment operation once, and $r1$ does not change when we update the relations *depositor*, *account*, *loan*, or *borrower*. In contrast, any modification we make to these relations changes the set of tuples in the view *all-customer* as well. Intuitively, at any given time, the set of tuples in the view relation is the result of evaluation of the query expression that defines the view at that time.

Thus, if a view relation is computed and stored, it may become out of date if the relations used to define it are modified. To avoid this, views are usually implemented as follows. When we define a view, the database system stores the definition of the view itself, rather than the result of evaluation of the relational-algebra expression that defines the view. Wherever a view relation appears in a query, it is replaced by the stored query expression. Thus, whenever we evaluate the query, the view relation gets recomputed.

Certain database systems allow view relations to be stored, but they make sure that, if the actual relations used in the view definition change, the view is kept up to date. Such views are called **materialized views**. The process of keeping the view up to date is called **view maintenance**, covered in Section 14.5. Applications that use a view frequently benefit from the use of materialized views, as do applications that demand fast response to certain view-based queries. Of course, the benefits to queries from the materialization of a view must be weighed against the storage costs and the added overhead for updates.

3.5.2 Updates through Views and Null Values

Although views are a useful tool for queries, they present serious problems if we express updates, insertions, or deletions with them. The difficulty is that a modification to the database expressed in terms of a view must be translated to a modification to the actual relations in the logical model of the database.

To illustrate the problem, consider a clerk who needs to see all loan data in the *loan* relation, except *loan-amount*. Let *loan-branch* be the view given to the clerk. We define this view as

create view *loan-branch* **as**
$$\Pi_{loan\text{-}number,\ branch\text{-}name}\ (loan)$$

Since we allow a view name to appear wherever a relation name is allowed, the clerk can write:

$$loan\text{-}branch \ \leftarrow\ loan\text{-}branch \cup \{(\text{L-37, “Perryridge”})\}$$

This insertion must be represented by an insertion into the relation *loan*, since *loan* is the actual relation from which the database system constructs the view *loan-branch*. However, to insert a tuple into *loan*, we must have some value for *amount*. There are two reasonable approaches to dealing with this insertion:

- Reject the insertion, and return an error message to the user.
- Insert a tuple (L-37, “Perryridge”, *null*) into the *loan* relation.

Another problem with modification of the database through views occurs with a view such as

create view *loan-info* **as**
$$\Pi_{customer\text{-}name,\ amount}(borrower \bowtie\ loan)$$

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500
null	*null*	1900

customer-name	*loan-number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17
Johnson	*null*

Figure 3.36 Tuples inserted into *loan* and *borrower*.

This view lists the loan amount for each loan that any customer of the bank has. Consider the following insertion through this view:

$$\textit{loan-info} \leftarrow \textit{loan-info} \cup \{(\text{“Johnson”}, 1900)\}$$

The only possible method of inserting tuples into the *borrower* and *loan* relations is to insert (“Johnson”, *null*) into *borrower* and (*null*, *null*, 1900) into *loan*. Then, we obtain the relations shown in Figure 3.36. However, this update does not have the desired effect, since the view relation *loan-info* still does *not* include the tuple (“Johnson”, 1900). Thus, there is no way to update the relations *borrower* and *loan* by using nulls to get the desired update on *loan-info*.

Because of problems such as these, modifications are generally not permitted on view relations, except in limited cases. Different database systems specify different conditions under which they permit updates on view relations; see the database system manuals for details. The general problem of database modification through views has been the subject of substantial research, and the bibliographic notes provide pointers to some of this research.

3.5.3 Views Defined by Using Other Views

In Section 3.5.1 we mentioned that view relations may appear in any place that a relation name may appear, except for restrictions on the use of views in update expressions. Thus, one view may be used in the expression defining another view. For example, we can define the view *perryridge-customer* as follows:

create view *perryridge-customer* **as**
$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name\,=\,\text{“Perryridge”}}\ (all\text{-}customer))$$

where *all-customer* is itself a view relation.

View expansion is one way to define the meaning of views defined in terms of other views. The procedure assumes that view definitions are not **recursive**; that is, no view is used in its own definition, whether directly, or indirectly through other view definitions. For example, if $v1$ is used in the definition of $v2$, $v2$ is used in the definition of $v3$, and $v3$ is used in the definition of $v1$, then each of $v1$, $v2$, and $v3$ is recursive. Recursive view definitions are useful in some situations, and we revisit them in the context of the Datalog language, in Section 5.2.

Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations. A view relation stands for the expression defining the view, and therefore a view relation can be replaced by the expression that defines it. If we modify an expression by replacing a view relation by the latter's definition, the resultant expression may still contain other view relations. Hence, view expansion of an expression repeats the replacement step as follows:

```
repeat
    Find any view relation $v_i$ in $e_1$
    Replace the view relation $v_i$ by the expression defining $v_i$
until no more view relations are present in $e_1$
```

As long as the view definitions are not recursive, this loop will terminate. Thus, an expression e containing view relations can be understood as the expression resulting from view expansion of e, which does not contain any view relations.

As an illustration of view expansion, consider the following expression:

$$\sigma_{customer\text{-}name=\text{“John”}}(\,perryridge\text{-}customer)$$

The view-expansion procedure initially generates

$$\sigma_{customer\text{-}name=\text{“John”}}(\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name\,=\,\text{“Perryridge”}}\ (all\text{-}customer)))$$

It then generates

$$\sigma_{customer\text{-}name=\text{“John”}}\ (\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name\,=\,\text{“Perryridge”}}\ (\Pi_{branch\text{-}name,\ customer\text{-}name}\ (depositor \bowtie\ account) \cup \Pi_{branch\text{-}name,\ customer\text{-}name}\ (borrower \bowtie\ loan))))$$

There are no more uses of view relations, and view expansion terminates.

3.6 The Tuple Relational Calculus

When we write a relational-algebra expression, we provide a sequence of procedures that generates the answer to our query. The tuple relational calculus, by contrast, is a **nonprocedural** query language. It describes the desired information without giving a specific procedure for obtaining that information.

A query in the tuple relational calculus is expressed as

$$\{t \mid P(t)\}$$

that is, it is the set of all tuples *t* such that predicate *P* is true for *t*. Following our earlier notation, we use $t[A]$ to denote the value of tuple *t* on attribute *A*, and we use $t \in r$ to denote that tuple *t* is in relation *r*.

Before we give a formal definition of the tuple relational calculus, we return to some of the queries for which we wrote relational-algebra expressions in Section 3.2.

3.6.1 Example Queries

Say that we want to find the *branch-name*, *loan-number*, and *amount* for loans of over $1200:

$$\{t \mid t \in loan \wedge t[amount] > 1200\}$$

Suppose that we want only the *loan-number* attribute, rather than all attributes of the *loan* relation. To write this query in the tuple relational calculus, we need to write an expression for a relation on the schema (*loan-number*). We need those tuples on (*loan-number*) such that there is a tuple in *loan* with the *amount* attribute > 1200. To express this request, we need the construct “there exists” from mathematical logic. The notation

$$\exists\, t \in r\ (Q(t))$$

means “there exists a tuple *t* in relation *r* such that predicate *Q*(*t*) is true.”

Using this notation, we can write the query “Find the loan number for each loan of an amount greater than $1200” as

$$\{t \mid \exists\, s \in loan\ (t[loan\text{-}number] = s[loan\text{-}number] \\ \wedge\ s[amount] > 1200)\}$$

In English, we read the preceding expression as “The set of all tuples *t* such that there exists a tuple *s* in relation *loan* for which the values of *t* and *s* for the *loan-number* attribute are equal, and the value of *s* for the *amount* attribute is greater than $1200.”

Tuple variable *t* is defined on only the *loan-number* attribute, since that is the only attribute having a condition specified for *t*. Thus, the result is a relation on (*loan-number*).

Consider the query “Find the names of all customers who have a loan from the Perryridge branch.” This query is slightly more complex than the previous queries, since it involves two relations: *borrower* and *loan*. As we shall see, however, all it requires is that we have two “there exists” clauses in our tuple-relational-calculus expression, connected by *and* ($\wedge$). We write the query as follows:

$$\begin{aligned}\{t \mid \exists\, s \in \mathit{borrower}\ (&t[\mathit{customer\text{-}name}] = s[\mathit{customer\text{-}name}]\\ &\wedge \exists\, u \in \mathit{loan}\ (u[\mathit{loan\text{-}number}] = s[\mathit{loan\text{-}number}]\\ &\qquad \wedge u[\mathit{branch\text{-}name}] = \text{“Perryridge”}))\}\end{aligned}$$

In English, this expression is “The set of all (*customer-name*) tuples for which the customer has a loan that is at the Perryridge branch.” Tuple variable *u* ensures that the customer is a borrower at the Perryridge branch. Tuple variable *s* is restricted to pertain to the same loan number as *s*. Figure 3.37 shows the result of this query.

To find all customers who have a loan, an account, or both at the bank, we used the union operation in the relational algebra. In the tuple relational calculus, we shall need two “there exists” clauses, connected by *or* (∨):

$$\begin{aligned}\{t \mid &\exists\, s \in \mathit{borrower}\ (t[\mathit{customer\text{-}name}] = s[\mathit{customer\text{-}name}])\\ &\vee \exists\, u \in \mathit{depositor}\ (t[\mathit{customer\text{-}name}] = u[\mathit{customer\text{-}name}])\}\end{aligned}$$

This expression gives us the set of all *customer-name* tuples for which at least one of the following holds:

- The *customer-name* appears in some tuple of the *borrower* relation as a borrower from the bank.
- The *customer-name* appears in some tuple of the *depositor* relation as a depositor of the bank.

If some customer has both a loan and an account at the bank, that customer appears only once in the result, because the mathematical definition of a set does not allow duplicate members. The result of this query appeared earlier in Figure 3.12.

If we now want *only* those customers who have *both* an account and a loan at the bank, all we need to do is to change the *or* (∨) to *and* (∧) in the preceding expression.

$$\begin{aligned}\{t \mid &\exists\, s \in \mathit{borrower}\ (t[\mathit{customer\text{-}name}] = s[\mathit{customer\text{-}name}])\\ &\wedge \exists\, u \in \mathit{depositor}\ (t[\mathit{customer\text{-}name}] = u[\mathit{customer\text{-}name}])\}\end{aligned}$$

The result of this query appeared in Figure 3.20.

Now consider the query “Find all customers who have an account at the bank but do not have a loan from the bank.” The tuple-relational-calculus expression for this query is similar to the expressions that we have just seen, except for the use of the *not* (¬) symbol:

$$\begin{aligned}\{t \mid &\exists\, u \in \mathit{depositor}\ (t[\mathit{customer\text{-}name}] = u[\mathit{customer\text{-}name}])\\ &\wedge \neg \exists\, s \in \mathit{borrower}\ (t[\mathit{customer\text{-}name}] = s[\mathit{customer\text{-}name}])\}\end{aligned}$$

customer-name
Adams
Hayes

Figure 3.37 Names of all customers who have a loan at the Perryridge branch.

This tuple-relational-calculus expression uses the $\exists\, u \ \in \ depositor\ (\ldots)$ clause to require that the customer have an account at the bank, and it uses the $\neg\ \exists\ s \ \in$ $borrower\ (\ldots)$ clause to eliminate those customers who appear in some tuple of the *borrower* relation as having a loan from the bank. The result of this query appeared in Figure 3.13.

The query that we shall consider next uses implication, denoted by $\Rightarrow$. The formula $P \ \Rightarrow \ Q$ means "P implies Q"; that is, "if P is true, then Q must be true." Note that $P \ \Rightarrow \ Q$ is logically equivalent to $\neg P \vee Q$. The use of implication rather than *not* and *or* often suggests a more intuitive interpretation of a query in English.

Consider the query that we used in Section 3.2.3 to illustrate the division operation: "Find all customers who have an account at all branches located in Brooklyn." To write this query in the tuple relational calculus, we introduce the "for all" construct, denoted by $\forall$. The notation

$$\forall\, t \ \in \ r\ (Q(t))$$

means "Q is true for all tuples t in relation r."

We write the expression for our query as follows:

$$\begin{aligned}
&\{t \mid \exists\, r \ \in \ customer\ (r[customer\text{-}name] = t[customer\text{-}name]) \wedge \\
&\quad (\ \forall\, u \ \in \ branch\ (u[branch\text{-}city] \ = \ \text{“ Brooklyn”} \ \Rightarrow \\
&\qquad\quad \exists\, s \ \in \ depositor\ (t[customer\text{-}name] \ = \ s[customer\text{-}name] \\
&\qquad\quad \wedge\, \exists\, w \ \in \ account\ (w[account\text{-}number] \ = \ s[account\text{-}number] \\
&\qquad\quad \wedge\, w[branch\text{-}name] \ = \ u[branch\text{-}name])))))\}
\end{aligned}$$

In English, we interpret this expression as "The set of all customers (that is, (*customer-name*) tuples *t*) such that, for *all* tuples *u* in the *branch* relation, if the value of *u* on attribute *branch-city* is Brooklyn, then the customer has an account at the branch whose name appears in the *branch-name* attribute of *u*."

Note that there is a subtlety in the above query: If there is no branch in Brooklyn, all customer names satisfy the condition. The first line of the query expression is critical in this case—without the condition

$$\exists\, r \ \in \ customer\ (r[customer\text{-}name] = t[customer\text{-}name])$$

if there is no branch in Brooklyn, any value of *t* (including values that are not customer names in the *depositor* relation) would qualify.

3.6.2 Formal Definition

We are now ready for a formal definition. A tuple-relational-calculus expression is of the form

$$\{t \mid P(t)\}$$

where *P* is a *formula*. Several tuple variables may appear in a formula. A tuple variable is said to be a *free variable* unless it is quantified by a $\exists$ or $\forall$. Thus, in

$$t \ \in \ loan \ \wedge \ \exists\, s \ \in \ customer(t[branch\text{-}name] \ = \ s[branch\text{-}name])$$

t is a free variable. Tuple variable *s* is said to be a *bound* variable.

A tuple-relational-calculus formula is built up out of *atoms*. An atom has one of the following forms:

- $s \in r$, where s is a tuple variable and r is a relation (we do not allow use of the $\notin$ operator)
- $s[x]\ \Theta\ u[y]$, where s and u are tuple variables, x is an attribute on which s is defined, y is an attribute on which u is defined, and Θ is a comparison operator ($<$, $\leq$, $=$, $\neq$, $>$, $\geq$); we require that attributes x and y have domains whose members can be compared by Θ
- $s[x]\ \Theta\ c$, where s is a tuple variable, x is an attribute on which s is defined, Θ is a comparison operator, and c is a constant in the domain of attribute x

We build up formulae from atoms by using the following rules:

- An atom is a formula.
- If P_1 is a formula, then so are $\neg P_1$ and (P_1).
- If P_1 and P_2 are formulae, then so are $P_1 \lor P_2$, $P_1 \land P_2$, and $P_1 \Rightarrow P_2$.
- If $P_1(s)$ is a formula containing a free tuple variable s, and r is a relation, then
$$\exists\, s \in r\ (P_1(s)) \text{ and } \forall\, s \in r\ (P_1(s))$$
are also formulae.

As we could for the relational algebra, we can write equivalent expressions that are not identical in appearance. In the tuple relational calculus, these equivalences include the following three rules:

1. $P_1 \land P_2$ is equivalent to $\neg\,(\neg(P_1) \lor \neg(P_2))$.
2. $\forall\, t \in r\ (P_1(t))$ is equivalent to $\neg\,\exists\, t \in r\ (\neg P_1(t))$.
3. $P_1 \Rightarrow P_2$ is equivalent to $\neg(P_1) \lor P_2$.

3.6.3 Safety of Expressions

There is one final issue to be addressed. A tuple-relational-calculus expression may generate an infinite relation. Suppose that we write the expression
$$\{t \mid \neg\,(t \in loan)\}$$
There are infinitely many tuples that are not in *loan*. Most of these tuples contain values that do not even appear in the database! Clearly, we do not wish to allow such expressions.

To help us define a restriction of the tuple relational calculus, we introduce the concept of the **domain** of a tuple relational formula, P. Intuitively, the domain of P, denoted $dom(P)$, is the set of all values referenced by P. They include values mentioned in P itself, as well as values that appear in a tuple of a relation mentioned in P. Thus, the domain of P is the set of all values that appear explicitly in

P or that appear in one or more relations whose names appear in P. For example, $dom(t \in loan \wedge t[amount] > 1200)$ is the set containing 1200 as well as the set of all values appearing in *loan*. Also, $dom(\neg\,(t \in loan))$ is the set of all values appearing in *loan*, since the relation $loan$ is mentioned in the expression.

We say that an expression $\{t \mid P(t)\}$ is *safe* if all values that appear in the result are values from $dom(P)$. The expression $\{t \mid \neg\,(t \in loan)\}$ is not safe. Note that $dom(\neg\,(t \in loan))$ is the set of all values appearing in *loan*. However, it is possible to have a tuple *t* not in *loan* that contains values that do not appear in *loan*. The other examples of tuple-relational-calculus expressions that we have written in this section are safe.

3.6.4 Expressive Power of Languages

The tuple relational calculus restricted to safe expressions is equivalent in expressive power to the basic relational algebra (with the operators $\cup, -, \times, \sigma$, and ρ, but without the extended relational operators such as generalized projection $\mathcal{G}$ and the outer-join operations) Thus, for every relational-algebra expression using only the basic operations, there is an equivalent expression in the tuple relational calculus, and for every tuple-relational-calculus expression, there is an equivalent relational-algebra expression. We will not prove this assertion here; the bibliographic notes contain references to the proof. Some parts of the proof are included in the exercises. We note that the tuple relational calculus does not have any equivalent of the aggregate operation, but it can be extended to support aggregation. Extending the tuple relational calculus to handle arithmetic expressions is straightforward.

3.7 The Domain Relational Calculus$**$

A second form of relational calculus, called **domain relational calculus**, uses *domain* variables that take on values from an attributes domain, rather than values for an entire tuple. The domain relational calculus, however, is closely related to the tuple relational calculus.

Domain relational calculus serves as the theoretical basis of the widely used QBE language, just as relational algebra serves as the basis for the SQL language.

3.7.1 Formal Definition

An expression in the domain relational calculus is of the form

$$\{< x_1,\ x_2, \ldots, x_n > \ \mid P(x_1,\ x_2, \ldots, x_n)\}$$

where $x_1,\ x_2, \ldots, x_n$ represent domain variables. *P* represents a formula composed of atoms, as was the case in the tuple relational calculus. An atom in the domain relational calculus has one of the following forms:

- $< x_1,\ x_2, \ldots, x_n > \in\ r$, where *r* is a relation on *n* attributes and $x_1,\ x_2, \ldots, x_n$ are domain variables or domain constants.

- $x \ \Theta \ y$, where x and y are domain variables and Θ is a comparison operator ($<$, $\leq$, $=$, $\neq$, $>$, $\geq$). We require that attributes x and y have domains that can be compared by Θ.
- $x \ \Theta \ c$, where x is a domain variable, Θ is a comparison operator, and c is a constant in the domain of the attribute for which x is a domain variable.

We build up formulae from atoms by using the following rules:

- An atom is a formula.
- If P_1 is a formula, then so are $\neg P_1$ and (P_1).
- If P_1 and P_2 are formulae, then so are $P_1 \ \vee \ P_2$, $P_1 \ \wedge \ P_2$, and $P_1 \ \Rightarrow \ P_2$.
- If $P_1(x)$ is a formula in x, where x is a domain variable, then

  $$\exists \ x \ (P_1(x)) \text{ and } \forall \ x \ (P_1(x))$$

  are also formulae.

As a notational shorthand, we write

$$\exists \ a, b, c \ (P(a, b, c))$$

for

$$\exists \ a \ (\exists \ b \ (\exists \ c \ (P(a, b, c))))$$

3.7.2 Example Queries

We now give domain-relational-calculus queries for the examples that we considered earlier. Note the similarity of these expressions and the corresponding tuple-relational-calculus expressions.

- Find the loan number, branch name, and amount for loans of over $1200:

  $$\{< l, b, a > \ | \ < l, b, a > \in \ loan \ \wedge \ a \ > \ 1200\}$$

- Find all loan numbers for loans with an amount greater than $1200:

  $$\{< l > \ | \ \exists \ b, a \ (< l, b, a > \in \ loan \ \wedge \ a \ > \ 1200)\}$$

Although the second query appears similar to the one that we wrote for the tuple relational calculus, there is an important difference. In the tuple calculus, when we write $\exists \ s$ for some tuple variable s, we bind it immediately to a relation by writing $\exists \ s \ \in \ r$. However, when we write $\exists \ b$ in the domain calculus, b refers not to a tuple, but rather to a domain value. Thus, the domain of variable b is unconstrained until the subformula $< \ l, b, a \ > \in \ \ loan$ constrains b to branch names that appear in the *loan* relation. For example,

- Find the names of all customers who have a loan from the Perryridge branch and find the loan amount:

$$\{< c, a > \mid \exists\ l\ (< c, l > \in\ borrower \\ \wedge \exists\ b\ (< l, b, a > \in\ loan\ \wedge\ b\ =\ \text{“Perryridge”}))\}$$

- Find the names of all customers who have a loan, an account, or both at the Perryridge branch:

$$\{< c > \mid \exists l\,(< c, l > \in\ borrower \\ \wedge \exists b, a\,(< l, b, a > \in\ loan\ \wedge\ b\ =\ \text{“Perryridge”})) \\ \vee \exists a\,(< c, a > \in\ depositor \\ \wedge \exists b, n\,(< a, b, n > \in\ account\ \wedge\ b\ =\ \text{“Perryridge”}))\}$$

- Find the names of all customers who have an account at all the branches located in Brooklyn:

$$\{< c > \mid \exists\, n\,(< c, n > \in\ customer) \wedge \\ \forall\, x, y, z\,(< x, y, z > \in\ branch\ \wedge\ y\ =\ \text{“Brooklyn”}\ \Rightarrow \\ \exists\, a, b\,(< a, x, b > \in\ account \wedge\ < c, a > \in\ depositor))\}$$

In English, we interpret this expression as “The set of all (*customer-name*) tuples c such that, for all (*branch-name*, *branch-city, assets*) tuples, x, y, z, if the branch city is Brooklyn, then the following is true”:

- ☐ There exists a tuple in the relation *account* with account number *a* and branch name *x*.
- ☐ There exists a tuple in the relation *depositor* with customer *c* and account number *a*.”

3.7.3 Safety of Expressions

We noted that, in the tuple relational calculus (Section 3.6), it is possible to write expressions that may generate an infinite relation. That led us to define *safety* for tuple-relational-calculus expressions. A similar situation arises for the domain relational calculus. An expression such as

$$\{< l, b, a > \ \mid \neg(< l, b, a > \in\ loan)\}$$

is unsafe, because it allows values in the result that are not in the domain of the expression.

For the domain relational calculus, we must be concerned also about the form of formulae within “there exists” and “for all” clauses. Consider the expression

$$\{< x > \ \mid \exists\, y\ (< x,\ y > \in\ r)\ \wedge\ \exists\, z\ (\neg(< x,\ z > \in\ r)\ \wedge\ P(x, z))\}$$

where P is some formula involving x and z. We can test the first part of the formula, $\exists\ y\ (< x,\ y > \in\ r)$, by considering only the values in *r*. However, to test the second part of the formula, $\exists\ z\ (\neg\ (< x,\ z > \in\ r)\ \wedge\ P(x, z))$, we must consider values for z that do not appear in *r*. Since all relations are finite, an infinite number of values do not appear in *r*. Thus, it is not possible, in general, to test the second part of the formula, without considering an infinite number of potential values for z. Instead, we add restrictions to prohibit expressions such as the preceding one.

In the tuple relational calculus, we restricted any existentially quantified variable to range over a specific relation. Since we did not do so in the domain calculus, we add rules to the definition of safety to deal with cases like our example. We say that an expression

$$\{< x_1,\ x_2, \ldots, x_n > \ | \ P\,(x_1,\ x_2, \ldots, x_n)\}$$

is safe if all of the following hold:

1. All values that appear in tuples of the expression are values from *dom*(*P*).
2. For every "there exists" subformula of the form $\exists\ x\ (P_1(x))$, the subformula is true if and only if there is a value *x* in $dom(P_1)$ such that $P_1(x)$ is true.
3. For every "for all" subformula of the form $\forall x\ (P_1(x))$, the subformula is true if and only if $P_1(x)$ is true for all values *x* from $dom(P_1)$.

The purpose of the additional rules is to ensure that we can test "for all" and "there exists" subformulae without having to test infinitely many possibilities. Consider the second rule in the definition of safety. For $\exists\ x\ (P_1(x))$ to be true, we need to find only one *x* for which $P_1(x)$ is true. In general, there would be infinitely many values to test. However, if the expression is safe, we know that we can restrict our attention to values from $dom(P_1)$. This restriction reduces to a finite number the tuples we must consider.

The situation for subformulae of the form $\forall x\ (P_1(x))$ is similar. To assert that $\forall x\ (P_1(x))$ is true, we must, in general, test all possible values, so we must examine infinitely many values. As before, if we know that the expression is safe, it is sufficient for us to test $P_1(x)$ for those values taken from $dom(P_1)$.

All the domain-relational-calculus expressions that we have written in the example queries of this section are safe.

3.7.4 Expressive Power of Languages

When the domain relational calculus is restricted to safe expressions, it is equivalent in expressive power to the tuple relational calculus restricted to safe expressions. Since we noted earlier that the restricted tuple relational calculus is equivalent to the relational algebra, all three of the following are equivalent:

- The basic relational algebra (without the extended relational algebra operations)
- The tuple relational calculus restricted to safe expressions
- The domain relational calculus restricted to safe expressions

We note that the domain relational calculus also does not have any equivalent of the aggregate operation, but it can be extended to support aggregation, and extending it to handle arithmatic expressions is straightforward.

3.8 Summary

- The **relational data model** is based on a collection of tables. The user of the database system may query these tables, insert new tuples, delete tuples, and update (modify) tuples. There are several languages for expressing these operations.
- The **relational algebra** defines a set of algebraic operations that operate on tables, and output tables as their results. These operations can be combined to get expressions that express desired queries. The algebra defines the basic operations used within relational query languages.
- The operations in relational algebra can be divided into
  - ☐ Basic operations
  - ☐ Additional operations that can be expressed in terms of the basic operations
  - ☐ Extended operations, some of which add further expressive power to relational algebra
- Databases can be modified by **insertion**, **deletion**, or **update** of tuples. We used the relational algebra with the **assignment operator** to express these modifications.
- Different users of a shared database may benefit from individualized **views** of the database. Views are "virtual relations" defined by a query expression. We evaluate queries involving views by replacing the view with the expression that defines the view.
- Views are useful mechanisms for simplifying database queries, but modification of the database through views may cause problems. Therefore, database systems severely restrict updates through views.
- For reasons of query-processing efficiency, a view may be **materialized**—that is, the query is evaluated and the result stored physically. When database relations are updated, the materialized view must be correspondingly updated.
- The **tuple relational calculus** and the **domain relational calculus** are non-procedural languages that represent the basic power required in a relational query language. The basic relational algebra is a procedural language that is equivalent in power to both forms of the relational calculus when they are restricted to safe expressions.
- The relational algebra and the relational calculi are terse, formal languages that are inappropriate for casual users of a database system. Commercial database systems, therefore, use languages with more "syntactic sugar." In Chap-

ters 4 and 5, we shall consider the three most influential languages: **SQL**, which is based on relational algebra, and **QBE** and **Datalog**, which are based on domain relational calculus.

Review Terms

- Table
- Relation
- Tuple variable
- Atomic domain
- Null value
- Database schema
- Database instance
- Relation schema
- Relation instance
- Keys
- Foreign key
  - ☐ Referencing relation
  - ☐ Referenced relation
- Schema diagram
- Query language
- Procedural language
- Nonprocedural language
- Relational algebra
- Relational algebra operations
  - ☐ Select σ
  - ☐ Project Π
  - ☐ Union $\cup$
  - ☐ Set difference $-$
  - ☐ Cartesian product $\times$
  - ☐ Rename ρ
- Additional operations
  - ☐ Set-intersection $\cap$
  - ☐ Natural-join $\bowtie$
  - ☐ Division $/$
- Assignment operation
- Extended relational-algebra operations
  - ☐ Generalized projection Π
  - ☐ Outer join
    - — Left outer join ⟕
    - — Right outer join ⟖
    - — Full outer join ⟗
  - ☐ Aggregation $\mathcal{G}$
- Multisets
- Grouping
- Null values
- Modification of the database
  - ☐ Deletion
  - ☐ Insertion
  - ☐ Updating
- Views
- View definition
- Materialized views
- View update
- View expansion
- Recursive views
- Tuple relational calculus
- Domain relational calculus
- Safety of expressions
- Expressive power of languages

Exercises

3.1 Design a relational database for a university registrar's office. The office maintains data about each class, including the instructor, the number of students enrolled, and the time and place of the class meetings. For each student−class pair, a grade is recorded.

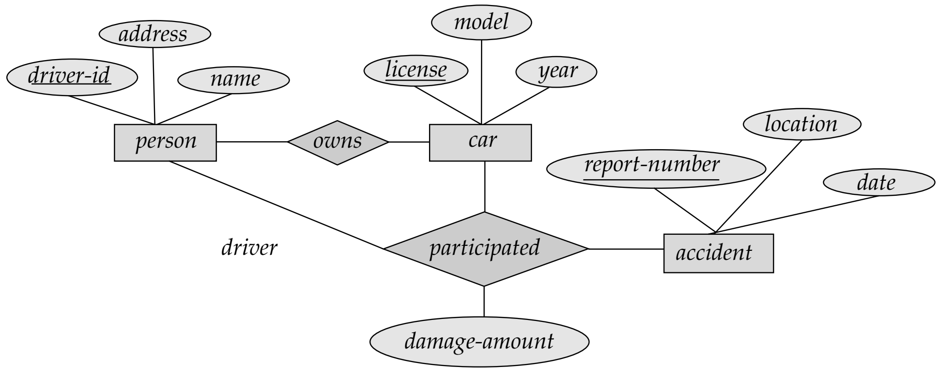


Figure 3.38 E-R diagram.

3.2 Describe the differences in meaning between the terms *relation* and *relation schema*. Illustrate your answer by referring to your solution to Exercise 3.1.

3.3 Design a relational database corresponding to the E-R diagram of Figure 3.38.

3.4 In Chapter 2, we saw how to represent many-to-many, many-to-one, one-to-many, and one-to-one relationship sets. Explain how primary keys help us to represent such relationship sets in the relational model.

3.5 Consider the relational database of Figure 3.39, where the primary keys are underlined. Give an expression in the relational algebra to express each of the following queries:

a. Find the names of all employees who work for First Bank Corporation.
b. Find the names and cities of residence of all employees who work for First Bank Corporation.
c. Find the names, street address, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000 per annum.
d. Find the names of all employees in this database who live in the same city as the company for which they work.
e. Find the names of all employees who live in the same city and on the same street as do their managers.
f. Find the names of all employees in this database who do not work for First Bank Corporation.
g. Find the names of all employees who earn more than every employee of Small Bank Corporation.
h. Assume the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.

3.6 Consider the relation of Figure 3.21, which shows the result of the query "Find the names of all customers who have a loan at the bank." Rewrite the query to include not only the name, but also the city of residence for each customer. Observe that now customer Jackson no longer appears in the result, even though Jackson does in fact have a loan from the bank.

employee (<u>*person-name*</u>, *street*, *city*)
works (<u>*person-name*</u>, *company-name*, *salary*)
company (<u>*company-name*</u>, *city*)
manages (<u>*person-name*</u>, *manager-name*)

Figure 3.39 Relational database for Exercises 3.5, 3.8 and 3.10.

a. Explain why Jackson does not appear in the result.
b. Suppose that you want Jackson to appear in the result. How would you modify the database to achieve this effect?
c. Again, suppose that you want Jackson to appear in the result. Write a query using an outer join that accomplishes this desire without your having to modify the database.

3.7 The outer-join operations extend the natural-join operation so that tuples from the participating relations are not lost in the result of the join. Describe how the theta join operation can be extended so that tuples from the left, right, or both relations are not lost from the result of a theta join.

3.8 Consider the relational database of Figure 3.39. Give an expression in the relational algebra for each request:

a. Modify the database so that Jones now lives in Newtown.
b. Give all employees of First Bank Corporation a 10 percent salary raise.
c. Give all managers in this database a 10 percent salary raise.
d. Give all managers in this database a 10 percent salary raise, unless the salary would be greater than $100,000. In such cases, give only a 3 percent raise.
e. Delete all tuples in the *works* relation for employees of Small Bank Corporation.

3.9 Using the bank example, write relational-algebra queries to find the accounts held by more than two customers in the following ways:

a. Using an aggregate function.
b. Without using any aggregate functions.

3.10 Consider the relational database of Figure 3.39. Give a relational-algebra expression for each of the following queries:

a. Find the company with the most employees.
b. Find the company with the smallest payroll.
c. Find those companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.

3.11 List two reasons why we may choose to define a view.

3.12 List two major problems with processing update operations expressed in terms of views.

3.13 Let the following relation schemas be given:

$$R \quad = (A, B, C)$$

$$S \;= (D, E, F)$$

Let relations *r*(*R*) and *s*(*S*) be given. Give an expression in the tuple relational calculus that is equivalent to each of the following:

a. $\Pi_A(r)$
b. $\sigma_{B\,=\,17}\,(r)$
c. $r \;\times\; s$
d. $\Pi_{A,F}\,(\sigma_{C\,=\,D}(r \;\times\; s))$

3.14 Let $R \;=\; (A,\; B,\; C)$, and let r_1 and r_2 both be relations on schema R. Give an expression in the domain relational calculus that is equivalent to each of the following:

a. $\Pi_A(r_1)$
b. $\sigma_{B\,=\,17}\,(r_1)$
c. $r_1 \;\cup\; r_2$
d. $r_1 \;\cap\; r_2$
e. $r_1 \;-\; r_2$
f. $\Pi_{A,B}(r_1) \bowtie \Pi_{B,C}(r_2)$

3.15 Repeat Exercise 3.5 using the tuple relational calculus and the domain relational calculus.

3.16 Let $R \;=\; (A,\; B)$ and $S \;=\; (A,\; C)$, and let $r(R)$ and $s(S)$ be relations. Write relational-algebra expressions equivalent to the following domain-relational-calculus expressions:

a. $\{< a > \;|\; \exists\, b\,(< a, b > \in\; r \;\wedge\; b \;=\; 17)\}$
b. $\{< a, b, c > \;|\; < a, b > \in\; r \wedge\; < a, c > \in\; s\}$
c. $\{< a > \;|\; \exists\, b\,(< a, b > \in\; r) \;\vee\; \forall\, c\,(\exists\, d\,(< d, c > \in\; s) \;\Rightarrow < a, c > \in\; s)\}$
d. $\{< a > \;|\; \exists\; c\;(< a, c > \in\; s \;\wedge\; \exists\; b_1, b_2\;(< a, b_1 > \in\; r \wedge\; < c, b_2 > \in\; r \;\wedge\; b_1 \;>\; b_2))\}$

3.17 Let $R \;=\; (A,\; B)$ and $S \;=\; (A,\; C)$, and let $r(R)$ and $s(S)$ be relations. Using the special constant *null*, write tuple-relational-calculus expressions equivalent to each of the following:

a. $r ⟖ s$
b. $r ⟗ s$
c. $r ⟕ s$

3.18 List two reasons why null values might be introduced into the database.

3.19 Certain systems allow *marked* nulls. A marked null $\perp_i$ is equal to itself, but if $i \;\neq\; j$, then $\perp_i \neq \perp_j$. One application of marked nulls is to allow certain updates through views. Consider the view *loan-info* (Section 3.5). Show how you can use marked nulls to allow the insertion of the tuple (“Johnson”, 1900) through *loan-info*.

Bibliographical Notes

E. F. Codd of the IBM San Jose Research Laboratory proposed the relational model in the late 1960s; Codd [1970]. This work led to the prestigious ACM Turing Award to Codd in 1981; Codd [1982].

After Codd published his original paper, several research projects were formed with the goal of constructing practical relational database systems, including System R at the IBM San Jose Research Laboratory, Ingres at the University of California at Berkeley, Query-by-Example at the IBM T. J. Watson Research Center, and the Peterlee Relational Test Vehicle (PRTV) at the IBM Scientific Center in Peterlee, United Kingdom. System R is discussed in Astrahan et al. [1976], Astrahan et al. [1979], and Chamberlin et al. [1981]. Ingres is discussed in Stonebraker [1980], Stonebraker [1986b], and Stonebraker et al. [1976]. Query-by-example is described in Zloof [1977]. PRTV is described in Todd [1976].

Many relational-database products are now commercially available. These include IBM's DB2, Ingres, Oracle, Sybase, Informix, and Microsoft SQL Server. Database products for personal computers include Microsoft Access, dBase, and FoxPro. Information about the products can be found in their respective manuals.

General discussion of the relational data model appears in most database texts. Atzeni and Antonellis [1993] and Maier [1983] are texts devoted exclusively to the relational data model. The original definition of relational algebra is in Codd [1970]; that of tuple relational calculus is in Codd [1972]. A formal proof of the equivalence of tuple relational calculus and relational algebra is in Codd [1972].

Several extensions to the relational calculus have been proposed. Klug [1982] and Escobar-Molano et al. [1993] describe extensions to scalar aggregate functions. Extensions to the relational model and discussions of incorporation of null values in the relational algebra (the RM/T model), as well as outer joins, are in Codd [1979]. Codd [1990] is a compendium of E. F. Codd's papers on the relational model. Outer joins are also discussed in Date [1993b]. The problem of updating relational databases through views is addressed by Bancilhon and Spyratos [1981], Cosmadakis and Papadimitriou [1984], Dayal and Bernstein [1978], and Langerak [1990]. Section 14.5 covers materialized view maintenance, and references to literature on view maintenance can be found at the end of that chapter.