PLUG-IN

T5 Designing Database Applications

LEARNING OUTCOMES

1. Describe the purpose of the relational database model in a database management system.
2. List the relational database model's basic components.
3. Describe why entities and attributes are organized into tables.
4. Describe how data redundancy is handled in the relational database model.
5. Explain the need for an entity-relationship diagram in a database management system.
6. Describe the Chen model symbols used in entity-relationship modeling.
7. Explain the purpose of normalization.
8. List the three normal forms typically used in normalization.

Introduction

Businesses rely on their database systems for accurate, up-to-date information. Without those databases of mission critical information, most businesses would be unable to perform their normal daily transactions, much less create summary reports that help management make strategic decisions. To be useful, the information must be accurate, complete, and organized in such a way that it can be retrieved when needed and in the format required.

The core units introduced the *database,* which maintains information about various types of objects (inventory), events (transactions), people (employees), and places (warehouses). A *database management system (DBMS)* is software through which users and application programs interact with a database. The *relational database model* is a type of database that stores its information in the form of logically related two-dimensional tables. This plug-in will build on the core units by providing specific details about how to design relational database applications.

Entities and Data Relationships

There are numerous elements in a business environment that need to store information, and those elements are related to one another in a variety of ways. Thus a

database must contain not only the information but also information about the relationships between the information.

The idea behind a database is that the user, either a person working interactively or an application program, has no need to worry about the way in which information is physically stored on disk. A database management system translates between the user's request for information and the physical storage.

A *data model* is a formal way to express data relationships to a database management system (DBMS). The underlying relationships in a database environment are independent of the data model and therefore independent of the DBMS that is being used. Before designing a database for any data model, data relationships need to be defined. An *entity-relationship diagram (ERD)* is a technique for documenting the relationships between entities in a database environment.

ENTITIES AND THEIR ATTRIBUTES

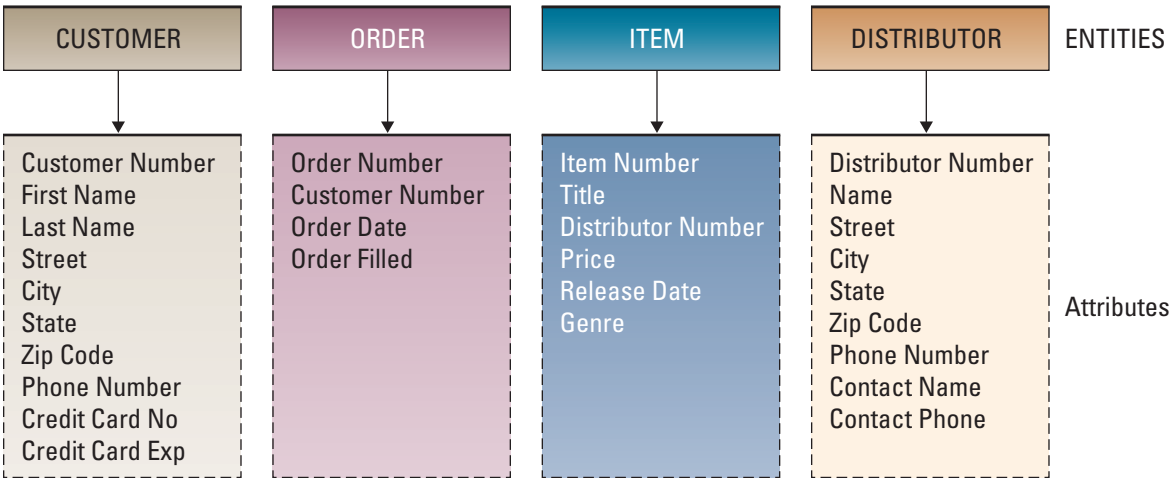
An *entity*, sometimes called a table, is a person, place, thing, transaction, or event about which information is stored. A customer is an entity, as is a merchandise item. Entities are not necessarily tangible; for instance, an appointment to see the doctor is an entity. *Attributes,* also called fields or columns, are characteristics or properties of an entity class. For example, a *CUSTOMER* entity can be described by a *Customer Number, First Name, Last Name, Street, City, State, Zip Code, Phone Number, Credit Card No,* and *Credit Card Exp* (refer to Figure T5.1).

When entities in a database are represented, only the attributes are stored. Each group of attributes models a single entity type in the real world, and values assigned to these attributes represent instances of objects (entity occurrences) corresponding to the entity. For example, in Figure T5.2, there are four instances of a *CUSTOMER* entity stored in a database. If there are 1,000 customers in the database, then there will be 1,000 instances of *CUSTOMER* entities. Instances can sometimes be referred to as records.

Entity Identifiers

An *entity identifier* ensures that each entity instance has a unique attribute value that distinguishes it from every other entity instance (an entity identifier is also referred to as a primary key, which will be discussed later in the plug-in). The primary purpose for entering the information that describes an entity into a database is to retrieve the information at some later date. This means there must be some way of distinguishing one entity from another in order to retrieve the correct entity. An entity identifier ensures that each entity has a unique attribute value that distinguishes it from every other entity.

FIGURE T5.1
Entities and Attributes Example



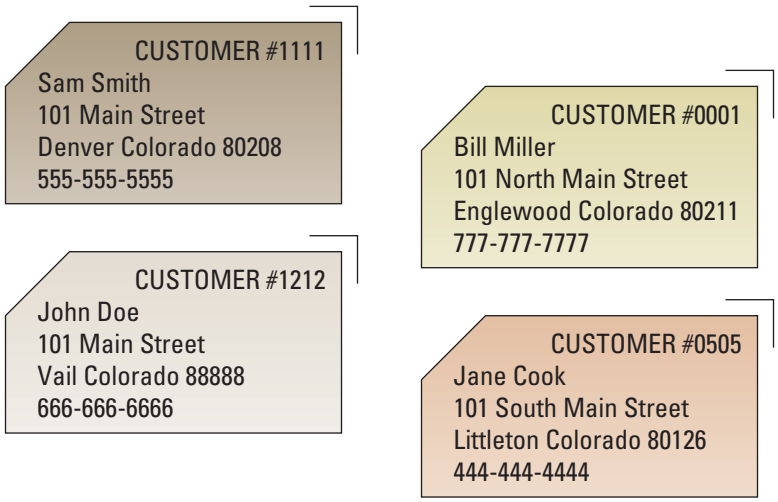


FIGURE T5.2
Customer Entity Instance

Assume, for example, that a local video store, Mega-Video, has two customers named John Smith. If an employee searches for the items John Smith has ordered, which John Smith will the DBMS retrieve? In this case, both of them. Since there is no way to distinguish between the two customers, the result of the query will be inaccurate. Mega-Video can solve the problem by creating an entity identifier.

Some entities, such as *ORDER*, come with natural identifiers, such as an *Order Number*. Typically, a unique, randomly generated number is assigned to entity identifiers.

A *constraint* is a rule to which some elements in a database must adhere. All entities must have a unique identifier that is a constraint. That is to say, when an instance of an entity in a database is stored, the DBMS needs to ensure that the new instance has a unique identifier. The enforcement of a variety of database constraints helps to maintain data consistency and accuracy.

ATTRIBUTES

There are several types of attributes, including:

- Simple versus composite.
- Single-valued versus multi-valued.
- Stored versus derived.
- Null-valued.

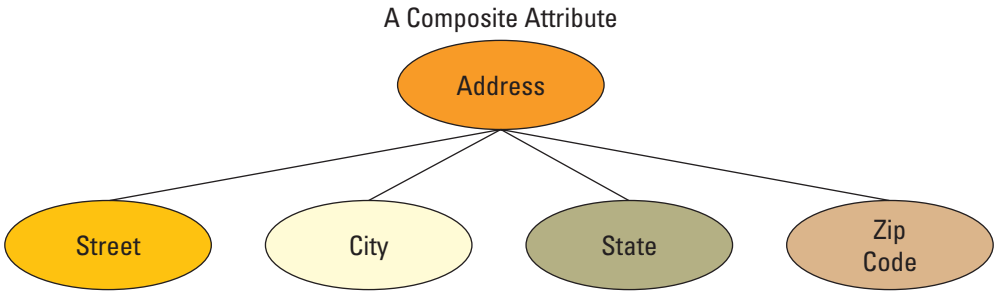
Simple versus Composite

Composite attributes can be divided into smaller subparts, which represent more basic attributes that have their own meanings. A common example of a composite attribute is *Address* (see Figure T5.3). *Address* can be broken down into a number of subparts, such as *Street, City, State, Zip Code. Street* may be further broken down by *Number, Street Name,* and *Apartment/Unit Number.* Attributes that are not divisible into subparts are called *simple attributes.*

Single-Valued versus Multi-Valued

When creating a relational database, the attributes in the data model must be single-valued. *Single-valued* means having only a single value of each attribute at any given time. For example, a *CUSTOMER* entity allows only one *Phone Number* for each

FIGURE T5.3
Composite Attributes



CUSTOMER. If a *CUSTOMER* has more than one *Phone Number* and wants them all included in the database, then the *CUSTOMER* entity cannot handle them.

The existence of more than one *Phone Number* turns the *Phone Number* attribute into a multi-valued attribute. *Multi-valued* means having the potential to contain more than one value for an attribute at any given time. An entity in a relational database cannot have multi-valued attributes. Those attributes must be handled by creating another entity to hold them.

In the case of the multiple *Phone Number*(s), a *PHONE NUMBER* entity needs to be created. Each instance of the entity would include the *Customer Number* of the person to whom the *Phone Number* belonged along with the *Phone Number*. If a customer had two *Phone Number*(s), then there would be two instances of the *PHONE NUMBER* entity for the *CUSTOMER* (see Figure T5.4).

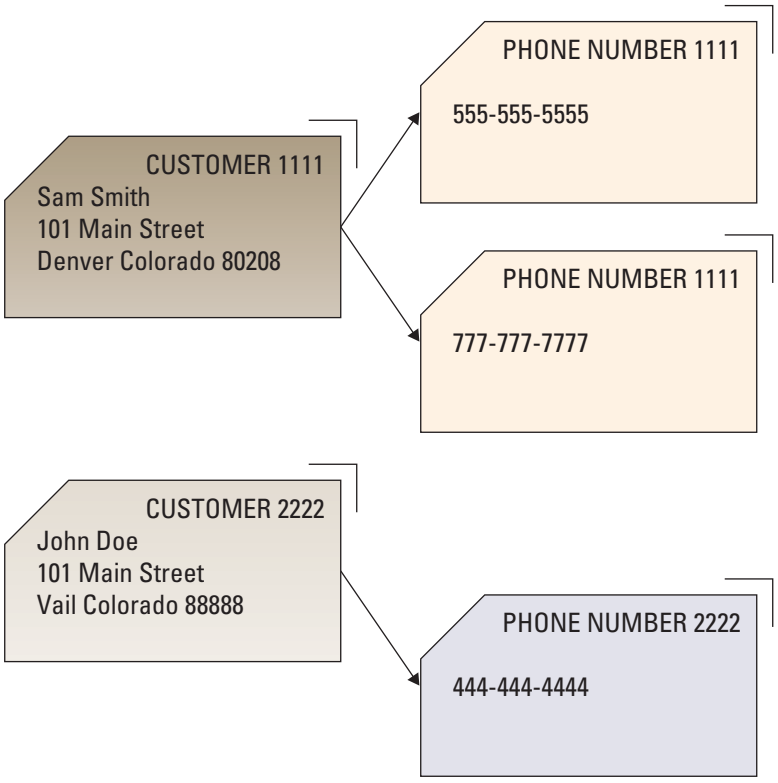


FIGURE T5.4
Customer Entity and Phone Number Entity

Multi-valued attributes can cause problems with the meaning of data in the database, significantly slow down searching, and place unnecessary restrictions on the amount of data that can be stored. Relational databases do not allow multi-valued attributes for this reason. For example, an *EMPLOYEE* entity with attributes for the *Name(s)* and *Birthdate(s)* of dependents would be considered multi-valued.

When searching a multi-valued attribute, a DBMS must search each value in the attribute, most likely scanning the contents of the attribute sequentially. A sequential search is the slowest type of search available.

Generally, a multi-valued attribute is a major hint that another entity is needed. The only way to handle multiple values of the same attribute is to create an entity for which multiple instances can be stored, one for each value of the attribute. In the case of the *EMPLOYEE* entity, a *DEPENDENT* entity that could be related to the *EMPLOYEE* entity needs to be created. There would be one occurrence of the *DEPENDENT* entity related to an occurrence of the *EMPLOYEE* entity for each of an employee's dependents. In this way, there is no limit to the number of an employee's dependents. In addition, each occurrence of the *DEPENDENT* entity would contain the *Name* and *Birthdate* of only one dependent, eliminating any confusion about which *Name* was associated with which *Birthdate,* as suggested in Figure T5.5. Searching would also be faster because the DBMS could use quicker search techniques on the individual *DEPENDENT* entity occurrences, without resorting to the slow sequential search.

Stored versus Derived

If an attribute can be calculated using the value of another attribute, it is called a *derived attribute*. The attribute that is used to derive the attribute is called a *stored attribute*. Derived attributes are not stored in the file, but can be derived when needed from the stored attributes. One example of a derived and stored attribute is a person's age. If the database has a stored attribute such as the person's *Date of Birth,* then you can create a derived attribute called *Age* from taking the *Current Date* (this is pulled from the system the database is running on) and subtracting the *Date of Birth* to get the age.

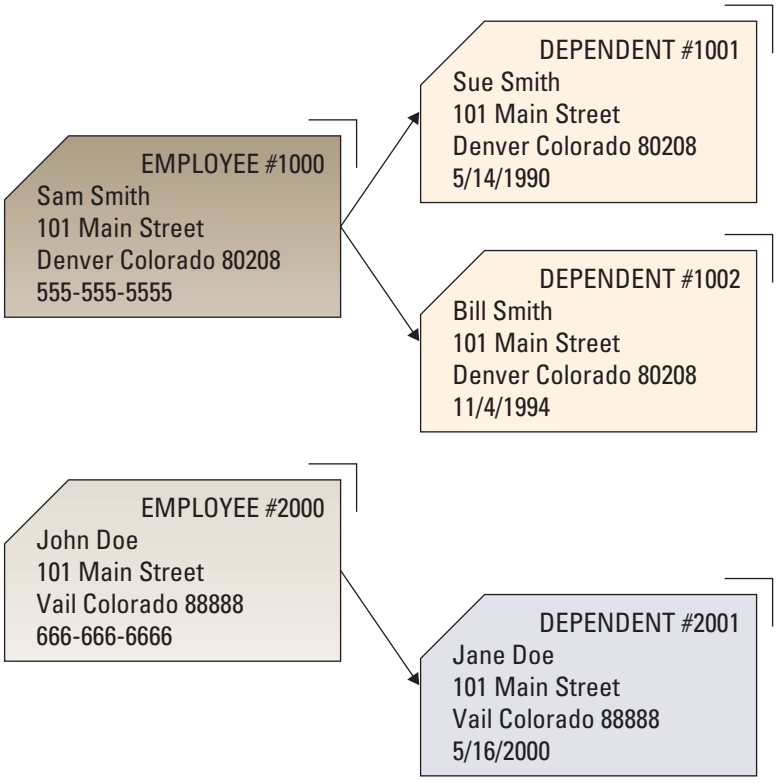


FIGURE T5.5
Employee Entity and Dependent Entity

Null-Valued

There are cases where an attribute does not have an applicable value for an attribute. For these situations, the *null-valued* attribute is created. A person who does not have a mobile phone would have null stored at the value for the *Mobile Phone Number* attribute. Null can also be used in situations where the attribute value is unknown. There are two cases where this can occur, one where it is known that the attribute is valued, but the value is missing, for example *Hair Color.* Every person has a hair color, but the information may be missing. Another situation is if *Mobile Phone Number* is null, it is not known if the person does not have a mobile phone or if that information is just missing.

Documenting Logical Data Relationships

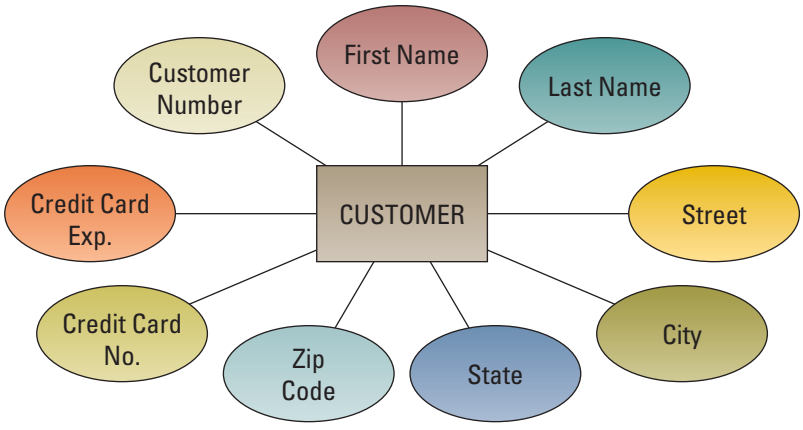
The two most commonly used styles of ERD notation are Chen, named after the originator of entity-relationship modeling, Dr. Peter Chen, and Information Engineering, which grew out of work by James Martin and Clive Finkelstein. It does not matter which is used, as long as everyone who is using the diagram understands the notation.

The Chen model uses rectangles to represent entities. Each entity's name appears in the rectangle and is expressed in the singular, as in *CUSTOMER*. The original Chen model did not provide a method for showing attributes on the ERD itself. However, many people have extended the model to include the attributes in ovals as illustrated in Figure T5.6.

BASIC DATA RELATIONSHIPS

The relationships that are stored in a database are between instances of entities. For example, a Mega-Video customer is related to the *ITEM*(s) he or she *ORDER*(s). Each instance of the *CUSTOMER* entity is related to instances of the specific *ITEM* ordered (see Figure T5.7). This is a purely conceptual representation of what is in the database and is completely unrelated to the physical storage of the data.

FIGURE T5.6
Chen Model with Attributes



When data relationships are documented, such as drawing an ERD, types of relationships among entities are shown, displaying the possible relationships that are allowable in the database. Unless a relationship is mandatory, there is no requirement that every instance of an entity be involved in the documented relationships. For example, Mega-Video could store information about a *CUSTOMER* without the customer having any current *ORDER*(s) to which it is related.

Once the basic entities and their attributes in a database environment have been defined, the next task is to identify the relationships among those entities. There are three basic types of relationships: (1) one-to-one, (2) one-to-many, and (3) many-to-many.

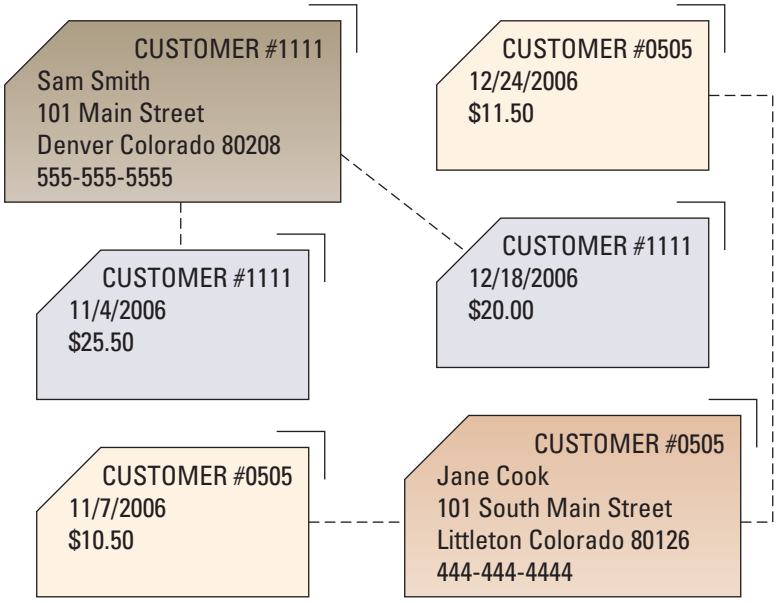


FIGURE T5.7
Entity Relationships

One-to-One

A *one-to-one (1:1)* relationship is between two entities in which an instance of entity A can be related to only one instance of entity B and entity B can be related to only one instance of entity A. Consider an airport in a small town and the town in which the airport is located, both of which are described in a database of small town airports (this would not be true for some major metropolitan cities, such as New York City with two major airports). Each of these might be represented as an instance of a different type of entity. As shown in Figure T5.8, the relationships between the two instances can then be expressed as "The airport is located in one and only one town and the town contains one and only one airport." The Chen method, as displayed in Figure T5.8, uses rectangles to document entities, a diamond to represent the relationship, and numbers to show the type of relationship in this example, (1:1).

This is a true one-to-one relationship because at no time can a single *AIRPORT* be related to more than one *TOWN* and no *TOWN* can be related to more than one *AIRPORT.* Although there are municipalities that have more than one *AIRPORT,* the *TOWN*(s) in this database are too small for that to happen.

True one-to-one relationships are rare in business. For example, assume that Mega-Video decides to start dealing with a new distributor of DVDs. At first, the company orders only one specialty title from the new distributor. The instance of the *DISTRIBUTOR* entity in the database is related to just the one merchandise *ITEM* instance. This would then appear to be a one-to-one relationship. Over time, Mega-Video may choose to order more titles from the new distributor, which would violate the rule that the distributor must be related to no more than one merchandise item. Therefore, this is not a true one-to-one relationship (this is an example of a one-to-many relationship, which is discussed next).

What if Mega-Video created a special *CREDIT CARD* entity to hold data about the credit cards that *CUSTOMER*(s) used to secure their rentals? Each *CUSTOMER* has only one credit card on file with the store. There would therefore seem to be a one-to-one relationship between the instance of a *CUSTOMER(s)* entity and the instance of the *CREDIT CARD* entity. In this case, it is a single entity. The *Credit Card Number,* the *Type of Credit Card,* and the *Credit Card Expiration Date* can all become attributes of the *CUSTOMER(s)* entity. Given that only one credit card is stored for each customer, the attributes are not multi-valued; no separate entity is needed.

FIGURE T5.8
A One-to-One Relationship

One-to-Many

A *one-to-many (1:M)* relationship is between two entities, in which an instance of entity A can be related to zero, one, or more instances of entity B and entity B can be related to only one instance of entity A. This is the most common type of relationship. In fact, most

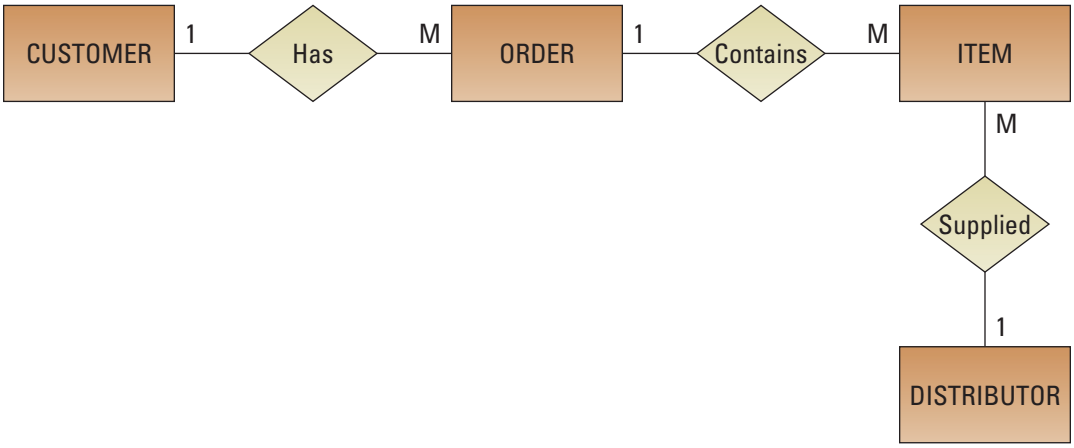


FIGURE T5.9
A One-to-Many Relationship

relational databases are constructed from the rare one-to-one relationship and numerous one-to-many relationships. Mega-Video typically *ORDER*(s) many *ITEM*(s) (in this scenario, an item is a DVD title) from each *DISTRIBUTOR* and a given *ITEM* comes from only one *DISTRIBUTOR* as Figure T5.9 demonstrates. Similarly, a *CUSTOMER* places many *ORDER*(s), but an *ORDER* comes from only one *CUSTOMER*.

When specifying data relationships, there needs to be an indication of the possible relationships, but an indication is not necessary that all instances of all entities participate in every documented relationship. There is no requirement that a *DISTRIBUTOR* be related to any merchandise *ITEM,* much less one or more merchandise *ITEM*(s). It might not make much sense to have a *DISTRIBUTOR* in the database from whom the company did not *ORDER,* but there is nothing to prevent data about that *DISTRIBUTOR* from being stored.

Many-to-Many

A *many-to-many (M:N)* relationship is between two entities in which an instance of entity A can be related to zero, one, or more instances of entity B and entity B can be related to zero, one, or more instances of entity A. There is a many-to-many relationship between a Mega-Video *CUSTOMER* and the merchandise *ITEMs* carried by the store (refer to Figure T5.10). A *CUSTOMER* can order many *ITEM*(s) and each *ITEM(s)* can be ordered from many *CUSTOMERs.*

Many-to-many relationships bring two major problems to a database's design. These issues and the way in which they are solved are discussed in the section "Dealing with Many-to-Many Relationships" below.

RELATIONSHIP CONNECTIVITY AND CARDINALITY

Cardinality expresses the specific number of entity occurrences associated with one occurrence of the related entity. In the Chen model, the cardinality is indicated by placing numbers beside the entities in the format of (x, y). The first number in the cardinality represents the minimum value and the second number stands for the maximum value.

The data relationships discussed thus far have defined those relationships by starting each with "zero," indicating that the cardinality in a given instance of an entity in a relationship is optional. Mega-Video can store data about a *CUSTOMER* in its database before the *CUSTOMER* places an *ORDER.* An instance of the *CUSTOMER* entity does not have to be related to any instances of the *ORDER* entity, meaning there is an *optional* cardinality.

FIGURE T5.10
A Many-to-Many Relationship

However, the reverse is not true for the Mega-Video database. An *ORDER must* be related to a *CUSTOMER.* Without a *CUSTOMER,* an *ORDER* cannot exist. As a

result, an *ORDER* is an example of a *weak entity,* one that cannot exist in the database unless a related instance of another entity is present and related to it. An instance of the *CUSTOMER* entity can be related to zero, one, or more orders. An instance of the *ORDER* entity must be related to one and only one *CUSTOMER,* having a cardinality of (1, 1). The "zero" option is not available to a weak entity. The relationship between an instance of the *ORDER* entity and the *CUSTOMER* is a mandatory relationship, as illustrated in Figure T5.11.

FIGURE T5.11
A Weak Entity and a Mandatory Relationship

Identifying weak entities and their associated mandatory relationships is important for maintaining the consistency and integrity of the database. Consider the effect of storing an *ORDER* without knowing the *CUSTOMER* to which it belongs. There would be no way to ship the *ITEM* to the *CUSTOMER,* causing a company to lose business.

There is a need to define the relationship between an *ORDER* and the *ORDER LINES* (the specific items on the order) as one-to-many because an *ORDER LINE* cannot exist in the database without its being related to an *ORDER.* An *ORDER LINE* is meaningless without knowing the *ORDER* to which it belongs.

In contrast, a merchandise *ITEM* can exist in a database without indicating the *DISTRIBUTOR* from which it comes (assuming that there is only one source per item). Data about a new *ITEM* can be stored before a *DISTRIBUTOR* is selected. In this case, the relationship between a *DISTRIBUTOR* and an *ITEM* is actually zero-to-many.

Documenting Relationships—The Chen Method

As briefly described earlier, the Chen method uses diamonds for relationships and lines to show the type of relationship between entities. Figure T5.12 displays the relationship between a Mega-Video *CUSTOMER* and an *ORDER.* The number "1" next to the *CUSTOMER* entity indicates that an *ORDER* belongs to at most one *CUSTOMER.* The letter "M" next to the *ORDER* entity indicates that a *CUSTOMER* can place one or more *ORDER*(s). The word within the relationship diamond gives some indication of the meaning of the relationship.

FIGURE T5.12
Chen Method Weak Entity Symbol

There is one major limitation to the Chen method of drawing ERDs—there is no obvious way to indicate weak entities and mandatory relationships. An *ORDER* should not exist in the database without a *CUSTOMER. ORDER* is a weak entity and its relationship with a *CUSTOMER* is mandatory.

Some database designers have added a new symbol to the Chen method for a weak entity, a double-bordered rectangle, as shown in Figure T5.13. Whenever a weak entity is introduced into an ERD, it indicates that the relationship between that entity and at least one of its parents is mandatory.

DEALING WITH MANY-TO-MANY RELATIONSHIPS

There are problems associated with many-to-many relationships. One problem is straightforward—the relational data model cannot handle many-to-many relationships directly; it is limited to one-to-one and one-to-many relationships. This means that the many-to-many relationships need to be replaced with a collection of one-to-many relationships in a relational DBMS.

FIGURE T5.13
Chen Method with Relationship

A second problem is a bit more subtle. To understand it, consider the relationship between an *ORDER* Mega-Video places with a *DISTRIBUTOR* and the merchandise *ITEM* on the *ORDER.* There is a many-to-many relationship between the *ORDER* and the *ITEM* because each *ORDER* can be for many *ITEM*(s) and, over time, each *ITEM* can appear on many *ORDER*(s). Whenever Mega-Video places an *ORDER* for an *ITEM,* the number of copies

of the *ITEM* varies, depending on the perceived demand for the *ITEM* at the time the *ORDER* is placed. Now the question: Where should we store the *Quantity* being ordered? It cannot be part of the *ORDER* entity because the *Quantity* depends on which item is being ordered. Similarly, the *Quantity* cannot be part of the *ITEM* entity because the *Quantity* depends on the specific *ORDER*.

Composite Entities

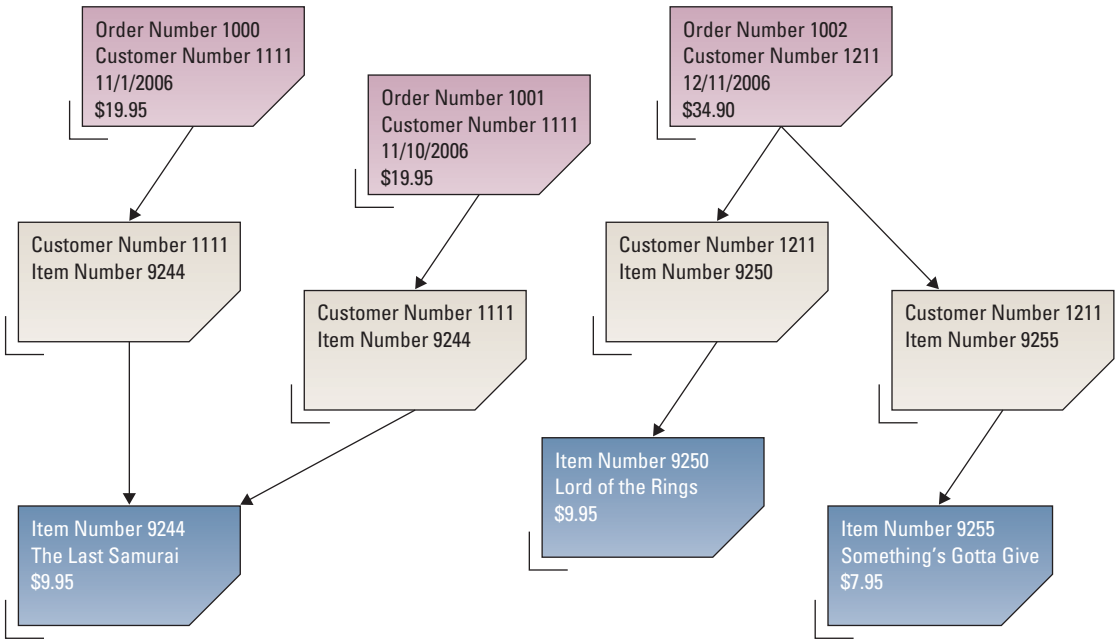
Entities that exist to represent the relationship between two other entities are known as *composite entities*. As an example of how composite entities work, consider the relationship between an *ORDER* placed by a *CUSTOMER* and the *ITEM*(s) in the *ORDER*. There is a many-to-many relationship between an *ITEM* and an *ORDER:* An *ORDER* can contain many *ITEM*(s) and over time, the same *ITEM* can appear on many *ORDER*(s).

What is needed is an entity that displays a specific title that appears on a specific order. Refer to Figure T5.14, there are three *ORDER* instances and three merchandise *ITEM* instances. The first *ORDER* for *Customer Number* 1111 (*Order Number* 1000) contains only one *ITEM* (*Item Number* 9244). The second *ORDER* for *Customer Number* 1111 (*Order Number* 1001) contains a second copy of *Item Number* 9244, but ordered on a different date. *Order Number* 1002, which belongs to *Customer Number* 1211, has two *ITEM*(s) in it (*Item Number* 9250 and *Item Number* 9255).

Therefore, a composite entity called *ORDER LINE* (think of it as a line item on a packing slip) is created to represent the relationship between an *ORDER* and a *PRODUCT*. Figure T5.15 demonstrates the Chen notation for ERDs; the symbol for a composite entity is the combination of a rectangle and a diamond.

Each *ORDER* is related to one *ORDER LINE* instance for each *ORDER* on which it appears. Each *ORDER LINE* instance is related to one and only one *ORDER;* it is also related to one and only one *PRODUCT* item. As a result, the relationship between an *ORDER* and its *ORDER LINE* is one-to-many (one order has one or more line items) and the relationship between an *ITEM* and the *ORDER* on which it appears is one-to-many (one item appears in zero, one, or more line items). The presence of the composite entity has removed the original many-to-many relationship and turned it into two one-to-many relationships.

FIGURE T5.14
Composite Entity Example



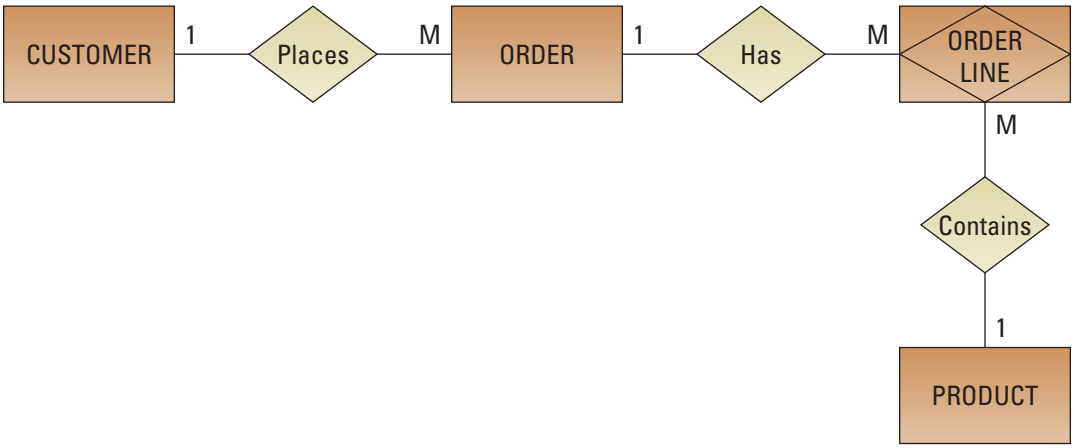


FIGURE T5.15
ERD of Composite Entity

SCHEMAS

A *schema* is a completed entity-relationship diagram representing the overall, logical plan of a database. This is the way in which the people responsible for maintaining the database will view the design. Users (both interactive users and application programs) may work with only a portion of the logical schema. In addition, both the logical schema and the users' views of the data are at the same time distinct from the physical storage.

The underlying physical storage, which is managed by the DBMS, is known as the *physical schema.* It is for the most part determined by the DBMS (only very large DBMSs give any control over physical storage). The benefit of this arrangement is that both database designers and users do not need to be concerned about physical storage, greatly simplifying access to the database and making it much easier to make modifications.

The Relational Data Model

Once the ERD is completed, it can be translated from a conceptual logical schema into the formal data model required by the DBMS. Most database installations are based on the relational data model.

The relational data model is the result of the work of one person, Edgar (E. F.) Codd. During the 1960s, Dr. Codd, trained as a mathematician, began working with existing data models. His experience led him to believe that these were clumsy and unnatural ways of representing data relationships. He therefore went back to mathematical set theory and focused on the construct known as a relation. Dr. Codd extended that concept to produce the relational database model, which he introduced in a historic seminal paper in 1970.

UNDERSTANDING RELATIONS

In mathematical set theory, a relation is the definition of a *table* with columns (e.g., attributes) and rows (e.g., records). The word "table" is used synonymously with "entity." The definition specifies what will be contained in each column of the table, but does not include information. When rows of information are included, an *instance* of a relation is created, such as the *CUSTOMER* relation in Figure T5.16.

At first glance, a relation looks much like a portion of a spreadsheet. Since it has its underpinnings in mathematical set theory, a relation has some very specific characteristics that distinguish it from other ways of looking at information. Each of these characteristics forms the basis of a constraint that will be enforced by the DBMS.

Customer			
Customer Number	**First Name**	**Last Name**	**Phone Number**
0001	Bill	Miller	777-777-7777
0505	Jane	Cook	444-444-4444
1111	Sam	Smith	555-555-5555
1212	John	Doe	666-666-6666

FIGURE T5.16
A Sample Customer Relation

Columns and Column Characteristics

Two or more tables within the same relational schema may have columns with the same names; in fact, in some circumstances, this is highly desirable. But a single table must have unique column names. When the same column name appears in more than one table and tables that contain that column are used in the same operation (e.g., query), the name of the column must be qualified by preceding it with the name of the table and a period, as in:

CUSTOMER.Customer Number, First Name, Last Name, Phone Number

Note the proper notation is to capitalize the table name (e.g., *CUSTOMER*) and all columns are in title case (Customer Number).

Rows and Row Characteristics

A row in a relation has the following properties:

- Only one value at the intersection of a column and row—a relation does not allow multi-valued attributes.
- Uniqueness—there are no duplicate rows in a relation.
- A primary key—a *primary key* is a field (or group of fields) that uniquely identifies a given entity in a table.

Primary Key

A *primary key* makes it possible to uniquely identify every row in a table. The primary key is important to define in order to retrieve every single piece of information put into a database.

As far as a relational database is concerned, there are only three pieces of information to retrieve for any specific bit of information: (1) the name of the table, (2) the name of the column, and (3) the primary key of the row. If primary keys are unique for every row, then the results will be exactly what was searched for. If they are not unique, then the data being retrieved will be a row with the primary key value, which may not be the row containing the data being searched.

The proper notation to use when documenting the name of the table, the column name, and primary key is as follows:

CUSTOMER(<u>Customer Number</u>, First Name, Last Name, Phone Number)

Again, notice that the table name is capitalized, the primary key is underlined, and it is the first attribute listed in the parenthetical statement containing the column names.

Along with being unique, a primary key must not contain the value null. *Null* is a special database value meaning "unknown." It is not the same as a zero or a blank. If one row has a null primary key, then the data structure is all right. The minute a second one is introduced, the property of uniqueness is lost. The presence of nulls in any primary key column is forbidden. This constraint, known as entity integrity, will be enforced by a DBMS whenever information is entered or modified. *Entity integrity* is a constraint on a relation that states that no part of a primary key can be null.

Selecting a primary key can be a challenge. Some entities have natural primary keys, such as purchase order numbers, as previously mentioned. Primary keys are often arbitrary, unique identifiers, such as a company attaches to the orders it sends to vendors. Two qualities of all primary keys are:

1. A primary key should contain some value that can never be null.
2. A primary key should never change.

REPRESENTING DATA RELATIONSHIPS

The use of identifiers in more than one relation was mentioned in the preceding section. This is the way in which relational databases represent relationships between entities.

Each table in Figure T5.17 is directly analogous to the entity by the same name in the Mega-Video ERD. The *CUSTOMER* entity is identified by a *Customer Number,* a randomly generated unique primary key. The *ORDER* entity is identified by an *Order Number,* another arbitrary unique primary key assigned by Mega-Video. The entity, *ORDER LINE,* tells the company which *ITEM*(s) are part of which *ORDER*. This table requires a *concatenated primary key* because multiple *ITEM*(s) can appear on multiple *ORDER*(s). The selection of this primary key, however, has more significance than simply identifying each row; it also represents a relationship between the *ORDER LINES,* the *ORDER* on which they appear, and the *ITEM*(s) being ordered. The *ITEM* entity is identified by an *Item Number,* an arbitrary unique primary key.

The *Item Number* column in the *ORDER LINE* relation is the same as the primary key of the *ITEM* table. This indicates a one-to-many relationship between the two tables. Similarly, there is also a one-to-many relationship between the *ORDER* and *ORDER LINE* tables because the *Order Number* column in the *ORDER LINE* table is the same as the primary key of the *ORDER* table.

When a table contains a column that is the same as the primary key of another table, the column is called a foreign key. A *foreign key* is a primary key of one table that appears as an attribute in another table and acts to provide a logical relationship between the two tables. The matching of foreign keys to primary keys represents data relationships in a relational database. As far as the user of a relational database is concerned, no structures show relationships other than the matching columns.

Foreign keys may be a part of a concatenated primary key or they may not be part of their table's primary key at all. Consider a pair of Mega-Video *CUSTOMER* and *ORDER* relations:

CUSTOMER(Customer Number, First Name, Last Name, Phone Number)

ORDER(Order Number, Customer Number, Order Date)

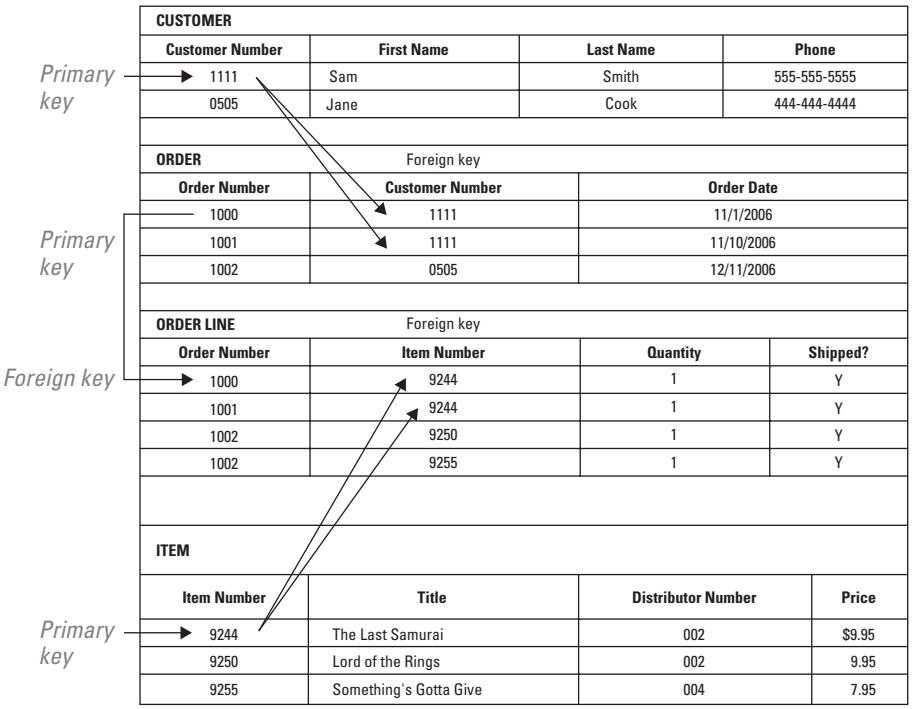


FIGURE T5.17
Relations from the Mega-Video Database

The *Customer Number* column in the *ORDER* table is a foreign key that matches the primary key of the *CUSTOMER* table. It represents the one-to-many relationship between *CUSTOMER*(s) and the *ORDER*(s) they place. However, the *Customer Number* is not part of the primary key of the *ORDER* table; it is a nonkey attribute that is nonetheless a foreign key, which is represented by using the double underline notation.

Technically, foreign keys need not have values unless they are part of a concatenated primary key; they can be null. However, in this particular database, Mega-Video would be in serious trouble if a *Customer Number* was null, since there would be no way to know which *CUSTOMER* placed an *ORDER*.

A relational DBMS uses the relationships indicated by matching data between primary and foreign keys. Assume that a Mega-Video employee wanted to see what *Titles* had been ordered with *Order Number* 1002. First, the DBMS identifies the rows in the *ORDER LINE* table that contain an *Order Number* of 1002. Then, it takes the *Item Number*(s) from the rows and matches them to the *Item Number*(s) in the *ITEM* table. In the rows where there are matches, the DBMS finally retrieves the associated *Title*.

Foreign Keys and Primary Keys in the Same Table

Foreign keys do not necessarily need to reference a primary key in a different table; they need only reference a primary key. As an example, consider the following employee relation:

EMPLOYEE(Employee Number, First Name, Last Name, Department, Manager Number)

A manager is also an employee. Therefore, the *Manager Number,* although named differently from the *Employee Number,* is actually a foreign key that references the primary key of its own table. The DBMS will therefore always ensure that whenever a user enters a *Manager Number,* that manager already exists in the table as an employee. Having a foreign key reference a primary key in the same table is relatively rare.

Referential Integrity

The procedure described in the preceding section works very well unless for some there is no *Order Number* in the *ORDER* table to match a row in the *ORDER LINE* table. This is undesirable since there is no way to ship the ordered *ITEM* because there is no way to find out which *CUSTOMER* placed the *ORDER*.

The relational data model enforces a constraint called *referential integrity,* which states that every non-null foreign key value must match an existing primary key value. Of all the constraints in a relational database, this is probably the most important because it ensures the consistency of the cross-references among tables.

Referential integrity constraints stored in the database are enforced automatically by the DBMS. As with all other constraints, each time a user enters or modifies data, the DBMS checks the constraints and verifies that they are met. If the constraints are violated, the data modification will not be allowed.

The Data Dictionary

The *data dictionary* is a file that stores definitions of information types, identifies the primary and foreign keys, and maintains the relationships among the tables. The structure of a relational database is stored in the database's data dictionary, or catalog. The data dictionary is made up of a set of relations, identical in properties to the relations used to hold information. No user can modify the data dictionary tables directly. Data manipulation language commands (e.g., Structured Query Language) that create and remove database structural elements work by modifying rows in data dictionary tables.

The following types of information are typically found in a data dictionary:

- Definitions of the columns that make up each table.
- Integrity constraints placed on relations.
- Security information (which user has the right to perform which operation of which table).

When a user attempts to access information in any way, a relational DBMS first goes to the data dictionary to determine whether the database elements the user has requested are actually part of the schema. In addition, the DBMS verifies that the user has the access rights to whatever he or she is requesting.

When a user attempts to modify information, the DBMS goes to the data dictionary to look for integrity constraints that may have been placed on the relation (see Figure T5.18). If the information has met the constraints, the modification is permitted. Otherwise, the DBMS returns an error message and does not allow the change. All access to a relational database is through the data dictionary.

RELATIONSHIPS AND BUSINESS RULES

In many ways, database design is as much an art as a science. The "correct" design for a specific business depends on the business rules; what is correct for one organization may not be correct for another.

Assume there is more than one store when creating a database for a retail establishment. One of the elements being modeled in the database is an employee's schedule. Before that can be done, the question of the relationship between an employee and a store needs to be answered: Is it one-to-many or many-to-many? Does an employee always work at only one store, in which case the relationship is one-to-many, or can an employee split his or her time between more than one store, producing a many-to-many relationship? This is not a matter of right or wrong database design, but an issue of how the business operates. These types of questions must be answered before you design a database.

Normalization

Normalization is the process of placing attributes into tables that avoid the problems associated with poor database design. Given any group of entities and attributes, there is a large number of ways to group them into relations.

There are at least two ways to approach normalization. The first is to work from an ERD. If the diagram is drawn correctly, then there are some simple rules to use to translate it into relations that will avoid most relational design problems. The drawback to this approach is that it can be difficult to determine whether the design is correct. The second approach is to use the theoretical concepts behind good design to create relations. This is a bit more difficult than working from an ERD, but often results in a better design.

NORMAL FORMS

Normal forms are the theoretical rules that the design of a relation must meet. Each normal form represents an increasingly stringent set of rules. Theoretically, the higher the normal form, the better the design of the relation.

As illustrated in Figure T5.19, there are six nested normal forms, indicating that if a relation is in one of the higher, inner normal forms, it is also in all of the normal forms surrounding it. In most cases, if relations are in third normal form (3NF), then most of the problems common to bad relational designs are avoided.

Table Name	Attribute Name	Contents	Type	Length	Format	Range	Req'd	Key	Referenced Table
CUSTOMER	Customer Number	Customer Number	VCHAR	10	X(10)		Y	PK	
	First Name	First Name	VCHAR2	12	X(12)		Y		
	Last Name	Last Name	VCHAR2	15	X(15)		Y		
	Street	Street Address	VCHAR2	20	X(20)		Y		
	City	City	VCHAR2	20	X(20)		Y		
	State	State	VCHAR2	2	X(2)		Y		
	Zip Code	ZIP Code	NUMBER	5	99999		Y		
	Credit Card No	Credit Card Number	NUMBER	15	X(15)		Y		
	Credit Card Exp	Credit Card Expiration Date	DATE	8	MM/DD/YYYY				
ORDER	Order Number	Order Number	NUMBER	5	99999	1-99999	Y	PK	
	Customer Number	Customer Number	VCHAR	10	X(10)		Y	FK	CUSTOMER
	Order Date	Order Date	DATE	8	MM/DD/YYYY		Y		
	Order Filled	Ordered Filled	DATE	8	MM/DD/YYYY		Y		
ORDER LINE	Order Number	Order Number	NUMBER	5	99999	1-99999	Y	FK	ORDER
	Item Number	Item Number	NUMBER	5	99999	1-99999	Y	FK	ITEM
	Quantity	Quantity	NUMBER	3	999	1-999	Y		
	Price	Selling Price	NUMBER	5	$999.99		Y		
	Shipped	Shipped	VCHAR2	1	X	Y/N	Y		
ITEM	Item Number	Item Number	Number	5	99999	1-99999	Y	PK	
	Title	Title	VCHAR2	25	X(25)		Y		
	Distributor	Distributor	VCHAR2	20	X(20)		Y		
	Price	Price	Number	5	$999.99		Y		

FIGURE T5.18

Data Dictionary Example

Boyce-Codd (BCNF) and fourth normal form (4NF) handle special situations that arise only occasionally. Fifth normal form (5NF) is a complex set of criteria that are extremely difficult to work with. It is very difficult to verify that a relation is in 5NF. Most practitioners do not bother with 5NF, knowing that if their relations are in 3NF (or 4NF if the situation warrants), then their designs are generally problem-free. BCNF, 4NF, and 5NF are beyond the scope of this plug-in; therefore they will not be discussed beyond what is mentioned in this section.

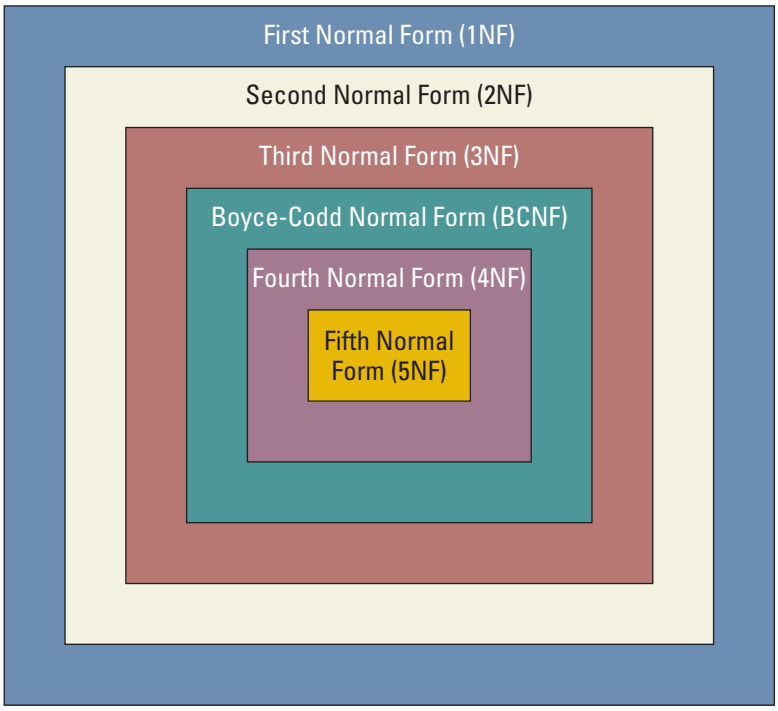


FIGURE T5.19
Normal Forms

First Normal Form (1NF)

First normal form (1NF) is where each field in a table contains different information. For example, in the column labeled "Customer," only customer names or numbers are permitted. A table is in first normal form (1NF) if the data are stored in a two-dimensional table with no repeating groups.

Although first normal form relations have no repeating groups, they are full of other problems. Expressed in the notation for relations that have been used in this plug-in, the relation notation would look like the following:

ORDER(Customer Number, First Name, Last Name, Street, City, State, ZIP, Phone, Order Number, Order Date, Item Number, Title, Price, Shipped)

The first thing is to determine the primary key for this table. The *Customer Number* alone will not be sufficient because the customer number repeats for every item ordered by the customer. The *Item Number* will also not suffice, because it is repeated for every order on which it appears. The *Order Number* cannot be used because it is repeated for every item on the order. The only solution is a concatenated key, in this example the combination of the *Order Number* and the *Item Number.*

Given that the primary key is made up of the *Order Number* and the *Item Number,* there are two important things that cannot be done with this relation:

- Data about a customer cannot be added until the customer places at least one order because without an order and an item on that order, there is no complete primary key.
- Data about a merchandise item cannot be added without that item being ordered. There must be an *Order Number* to complete the primary key.

First normal form relations can also present problems when deleting data. Consider, for example, what happens if a customer cancels the order of a single item:

- In cases where the deleted item was the only item on the order, all data about the order is lost.
- In cases where the order was the only order on which the item appeared, data about the item is lost.
- In cases where the deleted item was the only item ordered by a customer, all data about the customer is lost.

There is a final type of inconsistency in the *ORDER*(s) relation that is not related to the primary key: a modification, or update, anomaly. The *ORDER*(s) relation has

a great deal of unnecessary duplicated data; in particular, information about customers. When a customer moves, then the customer's data must be changed in every row, for every item on every order ever placed by the customer. If every row is not changed correctly, then data that should be the same are no longer the same.

Second Normal Form (2NF)

Second normal form (2NF) is when the relation is in first normal form and all nonkey attributes are functionally dependent on the entire primary key. The solution to anomalies in a first normal form relation is to break the relation down so that there is one relation for each entity in the 1NF relation. The *ORDER(s)* relation, for example, will break down into four relations (*CUSTOMER, ORDER, ORDER LINE,* and *ITEM*). Such relations are in at least 2NF.

Although second normal form eliminates problems from many relations, relations that are in second normal form still exhibit anomalies. Assume that each DVD title that Mega-Video carries comes from one *DISTRIBUTOR* and that each *DISTRIBUTOR* has only one warehouse, which has only one *Warehouse Phone Number.* The following relation is therefore in 2NF:

ITEM (Item Number, Title, Distributor, Warehouse Phone Number)

From each *Item Number,* there is only one value for the item's *Title, Distributor,* and *Warehouse Phone Number.* There is one insertion anomaly—data cannot be inserted about a *DISTRIBUTOR* until an item from the *DISTRIBUTOR* is entered. There is a deletion anomaly as well: if the only item from the *DISTRIBUTOR* is deleted, the data about the *DISTRIBUTOR* is lost.

Third Normal Form (3NF)

Third normal form (3NF) is when the relation is in second normal form and there are no transitive dependencies. In terms of entities, the items relation does contain two entities: the merchandise *ITEM* and the *DISTRIBUTOR.* The relation needs to be broken down into two smaller relations, both of which are now in 3NF:

ITEM(Item Number, Distributor Number)

DISTRIBUTOR(Distributor Number, Warehouse Phone Number)

NORMALIZED RELATIONS AND DATABASE PERFORMANCE

Normalizing the relations in a database separates entities into their own relations and makes it possible to enter, modify, and delete data without disturbing entities other than the one directly being modified. When relations are split so that relationships are represented by matching primary and foreign keys, DBMS is forced to perform matching operations between relations whenever a query requires data from more than one table. In a normalized database, data is stored about an *ORDER* in one relation, data about a *CUSTOMER* in a second relation, and data about the *ORDER LINE*(s) in yet a third relation. The operation typically used to bring the data into a single table to prepare an output, such as an *INVOICE,* is known as a join. A *join* is an operation that combines two relations by matching rows based on values in columns in the two tables. The matching relationship is usually primary key to foreign key.

In theory, a join looks for rows with matching values between two tables and creates a new row in a result table every time it finds a match. In practice, however, performing a join involves manipulating more data than the simple combination of the two tables being joined would suggest. Joins of large tables (those of more than a few hundred rows) can significantly slow down the performance of a DBMS.

PLUG-IN SUMMARY

A database management system, or DBMS, is considered a basic component of data processing. The main advantage of using a DBMS is to enforce a logical and structured organization of the data. Additionally, using a DBMS provides a central store of data that can be accessed by multiple users, from multiple locations. Data can be shared among multiple applications, instead of new iterations of the same data being reproduced and stored in new files for every new application.

The principal type of database used is a relational DBMS. Designing a database requires both a logical and physical design. The organization's data model should reflect its key business processes and decision-making requirements. Entity relationship diagrams and normalization are processes used to design a relational database.

MAKING BUSINESS DECISIONS

1. SportTech Events

SportTech Events puts on athletic events for local high school athletes. The company needs a database designed to keep track of the sponsor for the event and where the event is located. Each event needs a description, date, and cost. Separate costs are negotiated for each event. The company would also like to have a list of potential sponsors that includes each sponsor's contact information such as the name, phone number, and address. Each event will have a single sponsor, but a particular sponsor may sponsor more than one event. Each location will need an ID, contact person, and phone number. A particular event will use only one location, but a location may be used for multiple events. SportTech asks you to create an ERD from the information described above, and then create a normalization structure in 3NF.

2. Course and Student Schedules

Dick Scudder, the chairperson of the information technology department at the University of Denver, needs to create a database to keep track of all the courses offered by the department. In addition, Dick would like the database to include each instructor's basic contact information, such as ID number, name, office location, and phone number. Currently, Dick has nine instructors (seven full-time faculty members and two adjuncts) in the department.

For each course, Dick would like to keep track of the course ID, title, and number of credit hours. When courses are offered, the section of the course receives an ID number, and with that number, the department keeps track of which instructor is teaching the course.

Finally, Dick needs to be able to keep track of the IT students and to know which courses each student has taken. The information he would like to know about each student includes ID number, name, and phone number. He also needs to know what grade the student receives in each course.

Dick has asked you to create an ERD from the information described above, and then create a normalization structure in 3NF.

3. Foothills Athletics

Foothills Athletics is an athletic facility offering services in the greater Highlands Ranch, Colorado, area. All property owners living in Highlands Ranch are members of the Recreation

Function of the Highlands Ranch Community Association (HRCA). Foothills Athletics consists of a recreation facility where residents have the opportunity to participate in athletic activities, enroll their children in day camp or preschool, or participate in an HRCA program.

Personnel: Foothills Athletics has a number of employees, primarily fitness course instructors and administrative personnel (e.g., billing clerks, equipment managers, etc.). Records are kept on each employee, past and present, detailing employee name, address, phone number, date of hire, position, and status as either a current or former employee. Employees are assigned a unique four-digit Employee ID number when they are hired.

Members: When joining the Foothills Athletic center, individuals are assigned a unique four-digit Member ID number. This information along with their name, address, phone number, gender, birth date, and date of membership are recorded. At the time of enrollment, each member decides on one of three available membership types along with a fixed membership fee: Platinum ($400), Gold ($300), and Silver ($200). This is a one-time fee that establishes a lifetime membership.

Facilities and Equipment: Foothills Athletics has a variety of facilities and equipment choices. Each facility has a unique room number and a size limitation associated with it. Some of the rooms contain pieces of exercise equipment; all have a serial number (provided by its manufacturer) that is used for inventory purposes. In addition, for each piece of equipment, purchase date and the date of its last maintenance are recorded. Each piece of equipment belongs to a specific equipment type, such as stair master machine, and is assigned a unique three-digit identification number. The description, the manufacturer's model number, and the recommended maintenance interval for that model of equipment are also kept on file. Each equipment type is associated with a single manufacturer that is referenced by a unique two-digit manufacturer ID number. Additional information maintained on each manufacturer is the company name, address, and phone number.

The Task: You have been hired to assist Foothills Athletics with creating a database structure that will incorporate all the features and business rules mentioned above. You should start out developing an ERD and then proceed to create a normalization structure in 3NF.

4. On-the-Vine Vineyard

On-the-Vine Vineyard, Inc., is one of California's largest winemaking facilities in Sonoma Valley, striving to make both a visit to the vineyard and the wine tasting an unforgettable experience. On-the-Vine is a small, family-owned winery, specializing in limited production of premium quality Chardonnay, Sauvignon Blanc, Merlot, Syrah, Zinfandel, Sangiovese, Viognier, and Cabernet.

The Employees: On-the-Vine currently employs over 12 full-time employees, with positions ranging from administrative assistant to winemaker. Among the employees, supervisors have been appointed to manage the work of other employees. Each supervised employee reports to only one supervisor. Each employee, upon employment, is assigned a unique employee identification number. In addition to the employee's name, position, and identification number, the company also records each employee's Social Security number, address, phone number, and emergency contact.

The Vineyard: The grounds of On-the-Vine Vineyard include the Estate house with an award-winning rose garden, winery, and two vineyard plots of 40 acres each in separate locations. Each vineyard is managed by a single employee and is referred to by its own unique name, Sonoma Cellar and Sonoma Barrel. No employee manages more than one vineyard. Each vineyard is dedicated to the growing of a single grape variety per year.

As mentioned above, On-the-Vine Vineyard currently grows eight different grape varieties:

1. Chardonnay
2. Sauvignon Blanc
3. Merlot

4. Syrah
5. Zinfandel
6. Sangiovese
7. Viognier
8. Cabernet

The Winery: Each wine produced is given a unique identification number in addition to its name. Other information recorded for each wine is its vintage year, category (e.g., dry red, dessert, etc.), and percent alcohol, which is a legal requirement. Also recorded is the employee in charge of making that wine. Winemakers may be responsible for more than one wine at a time.

The composition of a wine may be entirely from a single grape variety or may be a blend of more than one variety. Several of the grape varieties are used in more than one blended wine.

The Customers: On-the-Vine customers are mainly restaurants and wine shops, but the winery also sells to individuals via the Internet. All customers are assigned a unique customer identification number, and this number is recorded along with their address and phone number. Individual customers also have their first name, last name, and date of birth, in order to demonstrate legal age, recorded. Restaurants and wine shops have their company name and tax identification number recorded.

All customers obtain their products by placing orders directly with On-the-Vine. Each order is assigned a unique order number, and the date the order is received, the product or products ordered, and the quantity or quantities desired are all recorded at the same time. A shipment status of "pending" is assigned to an order until it is actually shipped, whereupon the status is then changed to "shipped."

The Task: You have been hired to assist On-the-Vine Vineyard with creating a database structure that will incorporate all the features and business rules mentioned above. You should start out developing an ERD and then proceed to create a normalization structure in 3NF.