MICROSOFT EXAM OBJECTIVES COVERED IN THIS CHAPTER:

✓ Define entities. Considerations include entity composition and normalization.
  ▪ Specify entity attributes.
  ▪ Specify degree of normalization.

✓ Design entity keys. Considerations include FOREIGN KEY constraints, PRIMARY KEY constraints, and UNIQUE constraints.
  ▪ Specify attributes that uniquely identify records.
  ▪ Specify attributes that reference other entities.

✓ Design attribute domain integrity. Considerations include CHECK constraints, data types, and nullability.
  ▪ Specify scale and precision of allowable values for each attribute.
  ▪ Allow or prohibit NULL for each attribute.
  ▪ Specify allowable values for each attribute.
For many users, database design is a total mystery. Over the years, database management systems became easier to use and were included in office productivity tools. Databases were being created by people unaware of what a database design is. With a system like SQL Server, if the architecture of your database does not follow the rules of relational systems, you will end up with an unusable application.

In this chapter, we will discuss:

- Designing a database system
- The Entity/Relationship model
- The relational model and the normalization process
- The denormalization process

Designing a Database System

Whatever its size, the development of a database system may be split into five stages:

1. Planning and Analysis
2. Conceptual Design
3. Logical Design
4. Physical Design
5. Implementation
This chapter focuses on the first three phases of designing. Phase four is covered in Chapter 2. Phase 5 is discussed throughout the book, since it concerns the development of database objects.

The planning and analysis phase is an investigation phase, during which you are going to gather and analyze needed information. This stage is generally done with the help of users, and is crucial to the second phase.

You should involve users in the analysis phase because you do not know their job as well as they do, and because they should agree that what you are doing will work in the real world. You’ll probably encounter difficulties in involving users because they may not have time nor feel concerned. Insist! Explain to them that you are working for them and that the time they invest now with you will prevent lost time later due to an inadequate application. Sometimes, people won’t want to meet with you because they are intimidated; they fear to tell you that they dislike computers or fear you are going to use computer words or idioms they won’t understand. Users are involved only up to the logical design; they do not need to be concerned about DBMS systems or any computer related information.

The whole process of planning and analyzing information and building a conceptual design can be a long and costly one. That’s the reason why it’s often skipped, which is a huge a mistake! You can compare these two steps to designing a house. Would you think of building your house without blueprints? That’s the decision you make if you build a database without analysis and conceptual design. A deficient or even non-existent conceptual design leads to inaccurate logical design and an unusable physical one. Of course, we know the real world is not perfect. The borders between the analysis, conceptual, and logical designs are often blurred. You go from one stage to the other, back and forth. That’s why several methodologies or pieces of software will derive a conceptual design from a logical one, helping you to create your logical design step by step.

It’s always easier to modify the logical design than the physical one, once it has been implemented. Spend time creating your design! Check it! Make users validate it!
In fact, the conceptual and logical designs will generally be used as communication tools since they present data and functions in an understandable manner, even for the computer illiterate. The conceptual design is roughly made of two distinct models: the data model and the function model. The data model defines the data stored in the database; the function model defines the queries that will be executed on the database.

---

**Real World Scenario**

**The New Database Analysis**

You are a senior database developer of a medium-size organization and are called to analyze the future vacation and sick leave application for the Human Resources department. As an employee, you probably have ideas about information needed in this kind of application. But, as you are not working for HR, you do not know all of the subtleties of their jobs. The first step is to gather all necessary information, keeping in mind that even minor facts for you could be critical for someone in HR.

You make an appointment with Gary Pinkleton, the HR Manager, to determine the information the HR employees need. Fortunately, Gary is a well-organized guy, and he also invited Joan Winslow, the Office Manager, to the meeting. Each of them prepared a document summarizing the purpose of the application and the information that is needed. Unfortunately, they dislike computers, as do many of the HR employees, and you have to take that into account. They are paper and people oriented! You thank them for the good job they’ve done, and explain you would like to interview some HR employees in charge of managing vacation and sick leave, just to understand the way they work now. Then you will get back to them to discuss any issues met.

After gathering information through interviews, available documents, artifacts, etc., you have to analyze it. Probably the most important thing at that stage is to keep connected to the real world, being sure the analyzed information is representative of the situation. During the analysis stage, you have to organize, prioritize, and validate information.
Once you have all the necessary, accurate, and validated information, you can create cases to show actions between users and the new system and to describe the states of the system. Being able to identify these cases will help the conceptual and logical design because it will enhance the relationships between specific information.

After a couple of weeks, you meet again with Gary and Joan. You used Microsoft Visio to diagram your entity/relationship model and explain to them how you see things working. They are impressed by the simplicity of the diagram and the fact you clearly understood their need. They are reassured about the new application because you have not talked yet about computers or the way the application is to be implemented, but they can sense how it will work and see that all the necessary information is there.

Because you used what I call a user-oriented approach, they feel reassured about the new computer system and confident in the fact that the application will definitely help them do their jobs. On your side, you know that, as they participate in its design, they are partly responsible for the new system, so it will be easier to implement it in the department.

There is a classic confusion between the conceptual and the logical design. The ER model refers to the conceptual design stage and the relational model to the logical design stage. The ER model has been very popular because it is easy to derive it to create the relational model. Both models are discussed in the following sections.

The Entity/Relationship Model

Peter Chen first introduced the Entity/Relationship (ER) model in 1977. It has become very popular because an ER model is very simple to create and read, and can be used directly to create a relational model and transform its elements into database elements. The ER model translates
your analyzed information into data requirements, and, as stated earlier, is used to facilitate communications between the database architect and the future users of the new system. An ER model is made of three different elements:

- **Entity**, which represents real-world concepts, such as places, objects, events, persons, orders, customers, and so on.
- **Relationship**, which represents associations between objects, such as the fact that a customer may place an order.
- **Attribute**, which describes the entity, such as the invoice date or the customer first name.

In the next pages, you’ll notice there is a difference between an entity and an entity instance. An instance is an individual occurrence of an entity. In the relational world, an entity is equivalent to a table and an instance to a row.

Deriving entities, attributes, and relationships from the analysis phase may be an intricate process. What you need to do is to take every sentence of your conceptual model and transform the nouns (subjects) into the entities, the adjectives or nouns (direct objects) into the attributes, and the verbs into the relationships. Well, this may sound a little bit too easy, but in fact, that’s a logical process.

Let’s look at an example. The HR Manager of your company asked you to consider the following in your database (see previous design scenario sidebar):

- An employee is defined by his/her employee ID, first name, last name, hired date, and department.
- He/she applies for a vacation leave.

With these two statements, you discover two entities: Employee and Vacation Leave, plus five attributes of the Employee entity:

- ID
- First Name
- Last Name
- Hire Date
- Department
You also discover one relationship: applies for (between Employee and Vacation Leave). We do not have enough information to define what a Vacation Leave is, but that’s a kind of data we’ll need to gather from the HR Manager or any member of his/her team.

Let’s take a closer look at how to define entities and attributes first, then how to define relationships between entities.

**Defining Entities and Attributes**

**Microsoft Exam Objective**

- Define entities. Considerations include entity composition and normalization.
- Specify entity attributes.

As stated earlier, entities define real-word concepts, and attributes describe precisely these concepts. Peter Chen defines an entity as “a thing that can be distinctly identified.” There are two interesting aspects of this definition. First, he describes an entity as being a “thing.” It might be better to say an entity can be a thing, a concept, an object, an event, or a person, but on the whole, it is “something.” Second, he says that the entity can be distinctly identified. That may be the most important part of the concept. An item that does not have descriptive information and permits its identification is not an entity! So while analyzing a new database application, you should precisely describe and identify an item, so it has every chance to be an entity.

An attribute is a noun or an adjective that identifies or describes an entity. An attribute identifying an entity is called a *key attribute*. An attribute describing an entity is called a *non-key attribute*. For example, the employee ID is a key attribute of the employee entity. On the contrary, the employee’s first name is a non-key attribute. We’ll see later in this chapter that key attributes play an important role in relationships between entities.

Take the example of your address book. Each address represents a person or an organization you know—that’s an instance of the entity. Each address owns different attributes: the contact’s first name, last name, address, zip code, city, country, e-mail, phone number, and so on. If you have ever used Microsoft Excel to store that kind of data, you’ve used the spreadsheet format to create a table. An instance of the entity corresponds
to one row of this table and an attribute to one of its columns. From the interview conducted during the analysis phase, you can easily define entities and attributes from all the sentences and information gathered.

Generally, the consultant or anyone in charge of the analysis of the new database creates the entity/relationship diagram representing entities and relationships. In an ER model, each entity is represented by a labeled rectangle. The label is the name of the entity, which should always be a noun. Each entity attribute is listed inside the adequate entity rectangle.

Some ER gurus do not agree on listing the attribute directly on the ER model. In fact, there are different ways to represent entities, relationships, and attributes. The diagrams presented there conform to what is found in different Microsoft publications (official curriculum, books, white papers, and so on.) It may not exactly conform to Peter Chen’s ER historic representations, but it’s less academic and more understandable for a majority of people.

You can use Microsoft Visio 2000 to create an entity/relationship diagram, and to derive the logical and physical models from that point. Visio 2000 manages metadata directly to automatically generate tables, relationships, triggers, indexes, and so on from the diagrams. All the diagrams in this chapter have been made with Visio 2000 using the Source ER Model template, and all the examples are taken from the Pubs or Northwind databases shipped with SQL Server 2000.

To illustrate this concept of entities and attributes, let’s take a look at a part of the Northwind database, which is shipped with SQL Server 2000. While developing the Northwind database, the following have been extracted from the interview with the Purchase Manager of Northwind Traders Inc.:

- Every product is shipped by a specific supplier.
- We have the address, phone number, and fax number of every supplier. This is mandatory information because we must be able to contact them anytime.
- As far as the products are concerned, they are supplied by different suppliers, knowing that one supplier can supply many different products.

- For each product, we store its name, its price, the quantity per unit, the units on stock, and the units on order based on the reorder level, which is different for every product.

- Sometimes, we are forced to discontinue a product because it’s no longer produced or we cannot sell it anymore.

From these few statements, we discover two entities: products and suppliers, each of them having different attributes. Figure 1.1 illustrates entities and their associated attributes, plus the relationship.

**FIGURE 1.1** Entities/Relationship/Attributes

Using this kind of diagram, it becomes easy to communicate with users and have them help you validate your architectural choices. But, as you may have noticed, we find a relationship between both entities and key attributes. Let’s now take a look at how we define these keys and relationships.
Defining Relationships and Keys

Design entity keys. Considerations include FOREIGN KEY constraints, PRIMARY KEY constraints, and UNIQUE constraints.

- Specify attributes that uniquely identify records.
- Specify attributes that reference other entities.

The purpose of key attributes is to uniquely identify records and to allow relationships to be created between entities. SQL Server allows you to define keys and relationships in the physical model. These elements have to be identified early in the logical modeling process.

Relationships

Relationships are complex elements. They represent associations between entities and bind them with a set of defined rules. As stated earlier, relationships are generally derived from verbs or verb phrases in the conceptual model, but that's only the first step. Relationships carry three other main characteristics:

Direction indicates the source entity. For instance, a customer places an order, so the relationship goes from the customer entity to the order entity. The source of the relationship is often referred to as the parent entity and the destination as the child entity. In the preceding example, the customer entity is the parent and the order entity is the child. A relationship always goes from a parent to one or more children.

Cardinality defines the number of instances of a specific entity that could be associated with an instance(s) of another entity. For example, an employee can apply for one or more vacation leaves. An employee may apply for the first time (one); an older employee may have applied many times (many).

Existence determines the precedence between entities. That is, the entity that must exist before another entity is created. It may be optional or mandatory. For example, the relationship between a vacation leave and an employee is optional; the employee may apply for a vacation leave. But, the relationship between an employee and a department is mandatory: each employee belongs to one department.
A relationship is represented by a line between both entities. The type of line differs depending on the used methodology, the software, your university teacher, the country you live in, the weather. To be honest, there are as many notations as database experts. Let’s take three illustrated examples.

Figure 1.2 shows an arrow that indicates the direction of the relationships, labeled with its name and the cardinality on both sides.

**FIGURE 1.2** A relationship represented by a direction arrow

![Figure 1.2](image)

In this association, a supplier may supply many products, which is represented by the “supply” relationship. The character 0 (zero) on the supplier’s side indicates that a supplier can exist without related products. The character N (many) on the products side indicates that a supplier may supply many products. The direction of the arrow is natural and goes from one to many.

Figure 1.3 (used by default by Visio 2000) says that the line should be an arrow, with the arrowhead indicating the parent entity (the opposite of the “natural” direction), labeled with its name and cardinality on the child side.

**FIGURE 1.3** A default Visio 2000 relationship

![Figure 1.3](image)

In Figure 1.3, the “supply” relationship represents the same association as the preceding figure. The arrowhead indicates the parent entity, which is the source of the relationship. In fact, you should not see an arrowhead
but a starting point enlarging like a megaphone. The smallest side of the arrowhead indicates the “one” side, while the largest side indicates the “many” side. The asterisk character (\*) on the “many” side indicates the cardinality.

In Figure 1.4 (using crow’s feet) the vertical bar on the line indicates the “one” side of the relationship and a crow’s foot indicates the “many” side. The zero sign on the line indicates this is a one-to-zero-or-many relationship.

The different types of relationships are discussed later in this chapter.

**Figure 1.4** A relationship using a crow’s foot

In Figure 1.4, the “supply” relationship is always the same. The double vertical bar indicates the parent side. The first vertical bar next to the Suppliers entity indicates that a supplier must exist for every product (mandatory). The second vertical bar (representing a 1) indicates that one supplier (at most) must exist for every product. The crow’s foot next to the Products entity (representing many) indicates the child side. The 0 sign before the crow’s foot indicates that it is a one-to-zero-or-many relationship, meaning a supplier can be associated to zero, one, or many products.

On the physical side, SQL Server 2000 offers a diagram functionality that uses different notations. Figure 1.5 shows you the physical implementation of the above example.
As you can see, the relationship direction is illustrated by a key on the “one” side and an infinity sign (∞) on the “many” side. In the case of a one-to-one relationship, the key sign is on both sides, like in Figure 1.6.

In the above examples, you probably see that direction and existence are quite straightforward characteristics, which can be discovered easily. Cardinality is a little more complex, due to the different types of relationships: one-to-one, one-to-many, and many-to-many.

**One-to-One Relationship**

A one-to-one relationship (Figure 1.7) occurs when one instance of the parent entity is associated to one (at most) instance of the child entity. For instance, every company has only one CEO, and a CEO cannot be CEO of two different companies. It exists as a one-to-one relationship between the company entity and the CEO entity. In such a relationship, the direction is from the independent entity (the company) to the dependent entity (the CEO).
You may wonder what the use is of a one-to-one relationship. In this example, if there is only one CEO per company, why not create only one entity comprising all the necessary attributes? That is definitely the answer that can be given in a majority of cases. But you may decide to logically split information to keep entities small and manageable. This kind of relationship exists to take into account that some decisions are human and not only mathematical.

**One-to-Many Relationship**

A *one-to-many relationship* (the most frequently used relationship) occurs when one instance of the parent is associated to zero, one, or many instances of the child entity. For instance, a customer may place many orders. In this case, there is a one-to-many relationship between the customer entity and the order entity. The direction of a one-to-many relationship is always from the “one” side entity to the “many” side entity. Figure 1.8 shows a one-to-many relationship.

![Figure 1.8 A one-to-many relationship](image)

Figure 1.8 is equivalent to Figure 1.3. The asterisk represents the “many” side. In this example, each supplier supplies zero or many products.

**Many-to-Many Relationship**

A *many-to-many relationship* (Figure 1.9) occurs when one instance of the parent is associated with zero, one, or many instances of the child entity and when one instance of the child entity is associated with zero, one, or many instances of the parent entity. Even if the description may sound intricate, it’s quite a common situation. Consider when a customer places an order. He/she can order many products and those products can be on many orders. So, the relationship between the Orders entity and the Products entity is a many-to-many relationship. Many-to-many
relationships cannot be directly implemented in a relational database, but must be transformed into at least two one-to-many relationships, as we are going to see in the next sections. In a many-to-many relationship, the direction is arbitrary.

**FIGURE 1.9** A many-to-many relationship

![Diagram showing a many-to-many relationship](image)

Figure 1.9 shows a many-to-many relationship because an order contains one or many products and a product can be contained in zero, one, or many orders. That kind of relationship has to be resolved by inserting an entity called an association entity. Figure 1.10 shows a solution to our many-to-many relationship.

**FIGURE 1.10** A resolved many-to-many relationship

![Diagram showing a resolved many-to-many relationship](image)

By introducing the Order Details entity, we transform the many-to-many relationship into two one-to-many relationships. The new diagram shows that every order is made of one or many order details, and that each product may be referenced by zero, one, or many order details. As you can see, the original cardinality and existence are conserved by the new entity and relationships. A majority of many-to-many relationships are resolved that way.
**Recursive Relationship**

A recursive relationship is an epiphenomenon of a one-to-one or one-to-many relationship. A relationship is recursive when the source entity and the destination entity are the same. For example, every employee reports to his/her manager. But the manager is an employee, too. A recursive relationship is illustrated in Figure 1.11.

![Figure 1.11 A recursive relationship](image)

The previous figure shows that every employee reports to zero, one, or many employees. This kind of relationship is very easy to handle, since it is totally compatible with the relation model and SQL Server 2000.

**Keys**

Key attributes play a “key” role in relationships and in the relational model. There are two major types of keys: primary and foreign. Let’s take a look at what these keys are, what they are used for, and how they are chosen.

**Primary Key**

The *primary key* is an attribute or a set of attributes identifying unique instances of each entity. For example, the social security number identifies every citizen of a country, or the invoice number identifies every invoice created by a specific company. An entity may have multiple attributes or sets of attributes that identify unique instances of each entity. Each of these attributes or sets of attributes is called a *candidate key*. While an entity can have more that one candidate key, it has only one primary key. The other candidate keys are called *alternate keys*. 
If a key is made of multiple attributes, it is said to be composite.

Besides the fact of being an attribute or a set of attributes, a primary key must have the following properties to uniquely identify every instance:

- Every attribute must have a value. That means that no attribute composing the key can be NULL.
- The value of the key must be unique for every instance of the entity. If the key is composite, every group of attribute values has to be unique.

Some experts and gurus say that a primary key cannot be changed. In fact, even if it is not a good practice to permit the modification of a primary key, SQL Server 2000 permits it by default. You can forbid it by using triggers or stored procedure, as we’ll discover in the following pages.

The choice of the primary key may be complex and tricky, when no obvious choice is possible or when multiple choices are possible. Let’s look at two examples: an employee and a customer. An employee can be identified by different attributes: the combination of his/her first name and last name, his/her employee ID, or his/her social security number. In a small company, the combination of the first and last names could be a good choice, but in a medium or large company with thousands of employees this combination may not be unique. The social security number is a perfect choice for every company because every employee has one prior to his/her hiring. Now, the SSN may not be an identified or a necessary attribute, so having an employee ID automatically attributed by the system could be a good choice. Both attributes are candidate keys.

The customer can be identified by his/her ID or the combination of his/her name, address, and ZIP code, or you can create an increment ID to automatically identify the customer. The ID is not always known at record creation time, and the combination of name, address, and ZIP code creates quite a large key (that is containing too many attributes and too many characters). The last choice is sometimes called an artificial key because it has no real meaning to the entity, except being a unique identifier. The need for an artificial key arises when no attributes are really suitable or when the candidate keys seem too large.
SQL Server 2000 addresses the problem of artificial keys with identity property and UNIQUEIDENTIFIER datatypes. Read more about this in Chapter 3: Creating and Maintaining Tables.

In general, the primary key is identified in the ER by underlining the name of the attributes that compose the key and optionally listing it at the beginning of the attributes list (if other attributes are listed, of course). As you can see, Figure 1.12 is Figure 1.10 with the primary keys.

**FIGURE 1.12** Defining the primary keys

Note that the primary key of the associate entity (Order Details) is a composite key made of the primary keys of both parent entities. This is generally the case in this many-to-many relationship situation, though the primary key could be an artificial key, such as a counter.

SQL Server 2000 proposes to create a primary key through the primary key constraint, enforcing the non-NULL and unique properties of such a key. The creation of a primary key in SQL Server 2000 automatically creates an index. The physical creation of a primary key is discussed in Chapter 4: Implementing Data Integrity.

Primary keys are often noted as “PK” in diagrams. In SQL Server 2000, they are defined with a small yellow key. Every entity should have a primary key. As we see in a following section, this is a basic requirement for the first normal form.

Besides the primary key, the alternate keys can also be identified in the ER diagram and the relational model. An alternate key is a candidate key, so it may share the primary key characteristics: not NULL and uniqueness.
The alternate keys may be enforced in SQL Server 2000 using the unique constraints or unique indexes. The physical creation of a unique constraint is discussed in Chapter 4: Implementing Data Integrity, and the unique indexes are discussed in Chapter 5: Creating and Maintaining Indexes.

Besides the identifying entity instances, the primary key and eventually the alternate keys are used to define relationship source, linked to foreign keys.

**Foreign Key**

A *foreign key* is an attribute or a set of attributes that identifies the child side of a relationship. A foreign key is in fact the “migrating” primary key (or alternate key) of the parent entity. For example, if a customer entity is identified by a customer ID attribute, that customer ID attribute will be found in the order entity, since a relationship exists between customer and order. In Figure 1.13, the Orders entity is associated by one-to-many relationships with three different entities.

**FIGURE 1.13** Primary and foreign keys
The three non-key attributes of the Orders entity are “migrated” primary keys of the other entities. As you can see, discovering a foreign key is a straightforward process, once you know every primary key and every relationship.

A foreign key is linked to a primary or alternate key. In SQL Server 2000, a relationship is created through declarative integrity, with what is called a constraint. A relationship is created by declaring a foreign key constraint referencing either a primary key constraint or a unique constraint (alternate key) as the source.

To finish with relationships and foreign keys, the last notion is that of the “identifying relationship.” This is particularly useful if you use an ER design software like Visio 2000. A relationship is said to be identifying if the primary key of a child entity contains all the attributes of a foreign key. If the primary key of the child entity does not contain all the attributes of a foreign key, then the relationship is non-identifying.

In Visio 2000, as soon as you create an identifying relationship, the foreign key is automatically included in the primary key.

Figure 1.14 shows you an extract of the Entity/Relationship diagram of the Northwind database.

You may be used to more complicated or more complete diagrams due to the fact that only keys are listed here. Adding non-key attributes is a subject of discussion between experts. Some say that they should be included, other say they should not be. Depending on the complexity of your model, you may create different models or different levels allowing the display of non-key attributes.
Visio 2000 and SQL Server 2000 let you customize the display of your ER model so that you can declare every attribute but display only the ones necessary to your analysis.

Before switching to the relational model of our database, let’s spend some time with integrity. Integrity rules are essential to a database system, assuring that your data is correct and consistent.
Adding Data Integrity Rules

**Microsoft Exam Objective**

Design attribute domain integrity. Considerations include CHECK constraints, data types, and nullability.

- Specify scale and precision of allowable values for each attribute.
- Allow or prohibit NULL for each attribute.
- Specify allowable values for each attribute.

Integrity is one of the cornerstones of the relational model and has been over the years incorporated in every RDBMS (Relational Database Management System) on the market. There are four types of integrity:

- Domain integrity
- Entity integrity
- Referential integrity
- Enterprise integrity

**Domain Integrity**

A *domain* defines the possible values of an attribute. Domain integrity rules govern these values. In a database system, the *domain integrity* is defined by:

- The datatype and the length
- The NULL value acceptance
- The allowable values, through techniques like check constraints or rules
- The default value

For example, if you define that the attribute Age, of an Employee entity, is an integer, the value of every instance of that attribute must be numeric and an integer. If you define this attribute as always positive, then a negative value is forbidden. The value of this attribute being optional indicates that the attribute can be NULL. All these characteristics form the domain integrity of this attribute.

Datatypes in a database system can be numerous. Over the years, the storage need pushed RDBMS developers to introduce complex datatypes
to handle any case. Generally, datatypes can be divided into four types of attributes:

**Character**  *Character attributes* may have a fixed or a variable length, but the maximum length is precisely defined. For example, a ZIP code may be an attribute of five-character length.

**Numeric**  *Numeric attributes* can be integers of different lengths, or they can be real figures. In a computer, a numeric attribute can be two types of real figures: *floating point* and *fixed point*. For a floating point, the number of decimals is not known and the figure can be rounded to any decimal. For a fixed point, the architect defines the scale, which is the maximum number of decimals, and the precision, which is the maximum number of digits of the number. With these “precise” real figures, no rounding errors can occur. They are very useful for storing money values (for example, storing in the same entity values in dollars, Euros, and yen, up to the fourth decimal) or a precise decimal value.

Note that SQL Server 2000 proposes two “precise” real figures: numeric and decimal. Before SQL Server 7, their internal implementation was a little bit different. Since SQL Server 7, numeric and decimal figures are synonyms.

**Special**  *Special attributes* are, for example, datatypes like Boolean (true or false), GUID (Globally Unique Identifier), or Variant. They may be very useful for minimizing consumed space or providing special features.

We cover these special datatypes in detail in Chapter 3: Creating and Maintaining Tables.

**Binary**  *Binary attributes* can be anything besides character, numeric, and special types, such as a photograph, a sound, a file, a movie, and a binary string. These attributes are stored in the database in their binary format, without any modification. The RDBMS does not know what these binary data are, but knows they are a flow of binary digits.
The datatypes depend precisely on the RDBMS that you are going to use. But you can define in the conceptual model the global datatypes of every attribute, allowing you to define the domain integrity. For example, an attribute value can be implemented as one character allowing two values, Y and N, as a tiny integer allowing only 0 and 1, or as a bit, depending on the available features of your system. But you can define in the conceptual and logical model phases that this attribute has to be Boolean.

**Entity Integrity**

The *entity integrity* states that every instance of an entity has to be uniquely identified. The existence of the primary key is the core of the entity integrity. If you defined a primary key for each entity, they follow the entity integrity rule.

**Referential Integrity**

The *referential integrity* rules are enforced by the relationships between entities. As a starting point, the referential integrity rules state that a child instance cannot exist if there is no corresponding parent instance. For example, an order cannot exist without a matching customer, or an order detail cannot exist without the associated order.

Generally, referential integrity is defined by the following:

- You cannot delete a parent instance if one or many associated child instances exist.
- You cannot insert a child instance if the associated parent instance does not exist.

In other words: orphanage is impossible! Unfortunately, in the real world, orphans exist. Referential integrity defines rules to manage orphanage:

- Insert a child instance rule.
- Delete a child instance rule.
- Update a primary key rule.

**Insert Rules**

The insert rules include the following:

**Dependent** A child instance can be inserted only if a matching parent instance exists. This is generally the default rule.
Default  A child instance can always be inserted. If no matching parent exists, then the foreign key is set to the default value or to NULL.

Automatic  A child instance can always be inserted. If no matching parent exists, then one is created automatically.

No Effect  A child instance can always be inserted, even if no matching parent exists. This situation leads to no referential integrity and to data inconsistency!

Customized  A child instance can only be inserted if specific constraints are met. Depending on the existence of the matching parent instance, the custom function will follow the Dependent, the Default, the Automatic, or the No Effect rule.

Delete Rules
Delete rules include the following:

Restrict  A parent instance can be deleted if and only if no matching child instance exists. This is generally the default.

Cascade  The deletion of a parent instance triggers automatically the deletion of all matching child instances.

Default  The deletion of a parent instance triggers the update of the foreign key of all matching child instances to a default or a NULL value.

No Effect  A parent instance can always be deleted, regardless of the existence of child instances. This situation leads to no referential integrity and to data inconsistency!

Customized  A parent instance can only be deleted if specific constraints are met. Depending on the existence of the matching child instance(s), the custom function will follow the Cascade, the Default, or the No Effect rule.

Update Rules
Update rules include the following:

Restrict  A parent instance’s primary key cannot be updated if at least one child instance exists. This is generally the default rule.
Cascade    The update of a parent instance’s primary key triggers automatically the update of the foreign key of all matching child instances to the new value of the primary key.

Default    The update of a parent instance’s primary key triggers the update of the foreign key of all matching child instances to a default or a NULL value.

No Effect    A parent instance’s primary key can always be updated, regardless of the existence of child instances. This situation leads to no referential integrity and to data inconsistency!

Customized    A parent instance’s primary key can only be updated if specific constraints are met. Depending on the existence of the matching child instance(s), the custom function will follow the Cascade, the Default, or the No Effect rule.

In SQL Server 2000, only the Dependent insert rule, the Restrict or Cascade delete rules, and the Restrict or Cascade update rules can be enforced with foreign key and reference constraints.

Real World Scenario

Operation Order Issue

As a SQL Server freelance expert, you are called to design the new customer relationship management system of Golf Line Inc., a small company selling golf accessories through direct selling and the Internet. Martha Jarvis, the CEO, wants to know the company’s customers better. The golf players generally spend a lot of money on golf accessories, and she wants to be able to know who these people are, what they like and dislike, how much they spend every year, and so on.

You first meet Jon Albert, the in-house IT guy, who explains the different existing systems. The invoicing database is an old Access application, that slows down every day. So, you’ll need to incorporate invoicing facilities into the new system. The product database is managed by SQL Server. Every week, the in-house product manager receives new products from different suppliers, and decides with
Martha which products to add to their catalog and those to take out. You’ll need to use that product database in coordination with the new application.

After a quick meeting with Martha and Jon, you are hired to design and implement the new system. While designing it, you face classical problems of relationship rules. The first one deals with the Customer/Order relationship. You cannot create an order if matching customers does not exist, and you cannot delete a customer with matching orders.

You think about the Insert order situation. While entering a new order in the system, what happens if the customer does not exist? Sure, the front-end application will force the user to choose the customer first, but that situation could happen during batch inserts. So, the order is entered first and then the customer. If you decide to enforce the Dependant insert rule, the order cannot be inserted. With the Automatic insert rule, a new customer is automatically inserted, allowing the order to be inserted. The last operation is the update of this new customer.

Concerning the delete order, the problem may be a little more complex. Martha told you she wanted to mail people who have not ordered during the last six months, to be able to offer them special discounts and promotions. But at the same time, she told you to get rid of customers who have not been ordering for more than two years. She wants to keep a live database. The problem is simple: if you delete these customers, there will be inaccuracy in the orders, since the customer ID of these customers do not exist anymore. The Restrict delete rule does not work. If you implement the Cascade delete rule, you are going to lose every order the customers placed and paid. So you decide to implement a Customized delete rule: each time a customer is “deleted” for aging reasons, it is moved to an archive table, and the order is not impacted. This solution gives you the advantage of keeping a table of live customers and keeping all the information about the orders.

We all know that there are as many possible solutions to a problem as there are the number of people you are asking for a solution. These rules
are there to meet all these possible solutions. Depending on your knowledge of the skills of the architect, on the complexity of your solution, and on the software you are using, you’ll choose whatever solution suits you.

**Enterprise Integrity**

The last type of integrity is *enterprise integrity*, also called *business rules*. These rules, generally implemented through programmatic methods, like stored procedures or triggers on the database server side, define the way the company works. For example, you can state that a customer cannot place a new order if he still owes more than $10,000, or that an order greater than $200,000 has to be approved by the sales manager before being shipped. Enterprise integrity is generally not defined in the data model, but rather in the function model.

## The Relational Model and the Normalization Process

**Microsoft Exam Objective**

Define entities. Considerations include entity composition and normalization.

- Specify degree of normalization.

So far, we have discussed the conceptual model of our database, creating the ER model, entities, relationships, attributes, and attribute properties. It is now time to skip to the logical model, creating what is called the *relational model*. The relational model was first introduced by E.F. Codd in 1970, while he was a researcher at IBM. At that time, this model was revolutionary in the database world. In the relational model, two-dimensional tables represent data. Each table refers directly to an event, a person, and an object, like the entities we were talking about in the previous pages. In this model, a database is a collection of tables.

The organization of these tables is called the *logical model*, or logical view. The *physical model*, or physical view, is the real way data are stored in the database system that may differ from one software to another.
The physical model will be discussed in Chapter 2: Database Physical Modeling.

Going from the ER model to the relational model is very easy, since the first step is only a name change. Table 1.1 gives you the main differences between the main database elements, depending on the model or the formal names.

**Table 1.1** Name Differences of Database Elements

<table>
<thead>
<tr>
<th>ER Model</th>
<th>Relational Model</th>
<th>Formal Name</th>
<th>Physical Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td>Table</td>
<td>Relation</td>
<td>Table</td>
</tr>
<tr>
<td>Entity Instance</td>
<td>Row</td>
<td>Tuple</td>
<td>Record</td>
</tr>
<tr>
<td>Attribute</td>
<td>Column</td>
<td>Attribute</td>
<td>Field</td>
</tr>
</tbody>
</table>

No real formal representation of the logical model exists, except the one proposed by the ER model. So, you just transform entities in a table and attributes in a column, and the diagram remains the same. Let’s first take a look at the definition of the relational table.

**The Relational Table**

A *relational table* matches an ER entity. It defines the logical representation of the data and follows six rules:

Every column is atomic. This is definitely one important rule as far as relational tables are concerned. Being atomic means that a column contains only one value that cannot be broken into smaller pieces.

Atomicity examples are included in the section “First Normal Form” below.

Each column has a unique name. Each column matches an attribute, and must have a unique name within a table. Two different columns belonging to two different tables can have the same name.
Every value of a specific column is the same type. For the relational model, this rule means that every value of a column belongs to the same domain, and respects the domain integrity rule.

There are no duplicate rows. Each row is identified by a primary key, assuring its uniqueness. This rule states that every row can be accessed just by knowing its primary key.

The rows are unordered. The physical order of rows is meaningless. This property guarantees that the rows can be sorted in different ways, depending on what you need.

The columns are unordered. As with the rows, column order is meaningless. This property guarantees you can query the column of a table in the order you wish.

---

SQL Server 2000, like many other RDBMS, allows you to create tables without primary keys and with non-atomic columns. You can drive your car at 120 MPH downtown, but is it really a good idea? Concerning computer theory, I do not know a lot of things that have lasted more than 30 years, like the relational model. Therefore, it must be a good theory to still be the basis for RDBMS.

As you see, moving from the ER model to the relational model is straightforward if you just follow the previous rules. Nevertheless, while building our logical design, we did not really care about rows. If we start thinking about what happens when we “insert” data into the model, we may discover that we have duplicates, or information redundancy, which is information existing in more than one occurrence. That’s where the normalization process arrives. Normalizing data is the process of eliminating duplicated data by defining keys and creating new relationships and new entities.

Like ER modeling, the normalization process is mathematical and quite natural. A lot of database architects normalize their data without knowing the formal rules. Once you know them, you may find this process quite complex, but in fact, it’s straightforward if you use real-world data.

Each step of the normalization process starts with your logical model and ends with a new, normalized model. Each of these models has a name: First Normal Form, Second Normal Form, and so on. The model can
include up to five normal forms (and even six if we consider the Boyce-Codd Normal Form), but it’s been a common practice to stop at the Third Normal Form. In addition, the Microsoft Exam does not address normal forms beyond the third. In the following section, we will explain in detail how to get from a non-normalized model to the Third Normal Form and give you hints about the other three forms.

Normal Forms

*Normal form theory* is based on functional dependency between columns. Column A is said to be functionally dependent on column B if each value of B is associated with only one value of A. For example, an employee’s last name is functionally dependent on the employee’s ID. Knowing an ID, you are guaranteed to know the employee’s last name. In a relational table, every column must be dependent on the primary key. As you will see, this rule governs the normal forms.

Another concept is the *full functional dependency*. This concerns composite keys. Column A is said to be fully functionally dependent on B (B being a composite key) if A is functionally dependent on B and not on any subset of B. In other words, the whole primary key is necessary to accurately identify column A’s value. If this value can be identified accurately with only a few columns from the primary key, then A is not fully functionally dependent on the primary key.

Functional dependencies may be represented with the following notation:

\[ B \rightarrow A \]

This means A is functionally dependent on B, or knowing a value of B you know the matching value of A.

If A is functionally dependent on B, we also say that A is a *determinant* of B.

The goal of normal forms is to remove redundant data from relational tables by splitting the tables into smaller tables, without losing any data. It is necessary that the decomposition is lossless. That means that you can easily come back to the base table by combining the new created tables with a join.
First Normal Form

A relational table is in First Normal Form (1NF) if:

- It has a primary key.
- Each column is atomic.
- There is no repeating group of columns.

As you can see, the rules have nothing to do with redundancy, but almost follow some of the rules of relational tables. In fact, a table is said to be relational if it is in 1NF.

You should now understand the principle of the primary. So, let’s have a quick look at atomicity of columns. Imagine we create a table listing authors and the books they have written. This is shown in Figure 1.15.

**FIGURE 1.15** Non-atomic column

<table>
<thead>
<tr>
<th>au_id</th>
<th>Titles</th>
</tr>
</thead>
<tbody>
<tr>
<td>172-32-1176</td>
<td>Prolonged Data Deprivation: Four Case Studies</td>
</tr>
<tr>
<td>213-46-8915</td>
<td>The Busy Executive’s Database Guide; You Can Combat Computer Stress!</td>
</tr>
<tr>
<td>238-95-7766</td>
<td>But Is It User Friendly?</td>
</tr>
<tr>
<td>267-41-2394</td>
<td>Cooking with Computers: Surreptitious Balance Sheets; Sushi, Anyone?</td>
</tr>
<tr>
<td>274-80-9391</td>
<td>Straight Talk About Computers</td>
</tr>
<tr>
<td>341-22-1782</td>
<td>&lt;NULL&gt;</td>
</tr>
<tr>
<td>409-56-7008</td>
<td>The Busy Executive’s Database Guide</td>
</tr>
<tr>
<td>427-17-2319</td>
<td>Secrets of Silicon Valley</td>
</tr>
<tr>
<td>472-27-2349</td>
<td>Sushi, Anyone?</td>
</tr>
</tbody>
</table>

The Titles column can contain multiple values. For example, author 213-46-8915 wrote two books. He co-authored one of them with author 409-56-7008 (The Busy Executive’s Database Guide). It may become very difficult to query such a table and find information about a specific book. The first solution that comes to mind is to split the Titles column into two columns, as shown in Figure 1.16.
The solution addresses the issue of atomicity, but does not solve the query problem. It may be difficult, for example, to find if a specific title has been written by one or many authors, or to know the number of co-authors of one title. Worse, what if an author writes a third title? Where are you going to store it? Well, you could create a third Title column. But the problem would occur for the fourth, the fifth, and so on. Furthermore, even if you create 20 Title columns, it would be a waste of space for authors who only wrote one or two books.

If you want to put this table in 1NF, you could introduce a new column, title_id, identifying each book, and create a composite primary key (Figure 1.17).

The Relational Model and the Normalization Process
Now our table is in First Normal Form, since a primary key identifies every row, and every column is atomic. The problems we talked about are now solved: an author can write as many books he wishes, and it’s simple to group the table by title to list every co-author.

Let’s use a more complex table to uncover problems that could arise with a table in 1NF. The table in Figure 1.18 illustrates the entity described by the following:

- An author writes one or many books.
- Books are published by one publisher only.
- Books may be written by many authors, the royalties being shared amongst co-authors.
- Each publisher’s head office is in a particular city.
- Every publisher may publish one or more books.

**FIGURE 1.18** Royalties Table in First Normal Form

<table>
<thead>
<tr>
<th>au_id</th>
<th>title_id</th>
<th>royalty</th>
<th>pub_name</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>172-32-1176</td>
<td>PS3333</td>
<td>100</td>
<td>New Moon Books</td>
<td>Boston</td>
</tr>
<tr>
<td>213-46-8915</td>
<td>BU1032</td>
<td>40</td>
<td>Algodata Infosystems</td>
<td>Berkeley</td>
</tr>
<tr>
<td>213-46-8915</td>
<td>BU2075</td>
<td>100</td>
<td>New Moon Books</td>
<td>Boston</td>
</tr>
<tr>
<td>238-95-7766</td>
<td>PC1035</td>
<td>100</td>
<td>Algodata Infosystems</td>
<td>Berkeley</td>
</tr>
<tr>
<td>267-41-2394</td>
<td>BU1111</td>
<td>40</td>
<td>Algodata Infosystems</td>
<td>Berkeley</td>
</tr>
<tr>
<td>267-41-2394</td>
<td>TC7777</td>
<td>30</td>
<td>Binnet &amp; Hardley</td>
<td>Washington</td>
</tr>
<tr>
<td>274-80-9931</td>
<td>BU1032</td>
<td>100</td>
<td>Algodata Infosystems</td>
<td>Berkeley</td>
</tr>
<tr>
<td>409-56-7008</td>
<td>BU1032</td>
<td>60</td>
<td>Algodata Infosystems</td>
<td>Berkeley</td>
</tr>
<tr>
<td>427-17-2319</td>
<td>PC8888</td>
<td>50</td>
<td>Algodata Infosystems</td>
<td>Berkeley</td>
</tr>
<tr>
<td>472-27-3499</td>
<td>TC7777</td>
<td>30</td>
<td>Binnet &amp; Hardley</td>
<td>Washington</td>
</tr>
</tbody>
</table>

The Royalties relational table, shown in Figure 1.18, is already in First Normal Form. Nevertheless, it contains redundant data. For example, the publisher_name or the city is repeated. Redundancy may cause anomalies during data insertion, deletion, or update. For example:

- You cannot insert a new publisher until it has published at least one book.
- If you delete a row, you are deleting information about an author and a book, and you lose information about the publisher.
If you update the city of a publisher, you have to update every row of the author who has been published by this publisher.

We have to decompose this table to achieve Second Normal Form.

**Second Normal Form**

A relational table is in Second Normal Form (2NF) if:

- It is in 1NF.
- Every non-key column is fully functionally dependent on the primary key.

In Figure 1.18, the Royalties table is in 1NF but not in 2NF because the columns title and publisher_id depend only on the title_id and not on the key (au_id, title_id). You can easily establish this fact if you study the functional dependencies of the table:

\[(au_id, title_id) \rightarrow royalty\]
\[title_id \rightarrow pub\_name, city\]
\[pub\_name \rightarrow city\]

So, two non-key columns are not fully functionally dependent on the primary key. That is, they do not depend on the entire primary key, but only on one of its subsets. Decomposing a table in 1NF to achieve 2NF is a logical process:

1. Identify all the determinant parts of the primary key and their dependant columns.
2. Create a new table from every determinant and their dependant columns.
3. The determinant becomes the primary key of the new table.
4. Delete the dependant columns from the source table. Do not delete the determinant, since it will become the foreign key.

You may rename the source table if you wish to keep meaningful information. To transform the Royalties table to 2NF, we create a new table, named Titles, with the columns title_id, pub_name, and city. Title_id becomes the primary key of this new table (Figure 1.19).
Though the tables are in 2NF, update anomalies can still occur. For example:

- You cannot insert a new publisher if you do not know the title_id of at least one of the books published.

- If you delete a row in the Titles table, you lose the information about the publisher at the same time. A publisher may disappear if you delete its last published book referenced in the table.

To avoid these anomalies, the Titles table should be decomposed to achieve the Third Normal Form.

**Third Normal Form**

A relational table is in Third Normal Form (3NF) if:

- It is in 2NF.
Every non-key column is functionally dependent only on the primary key. In other words, a non-key column cannot be dependent on another non-key column.

In our example, the Royalties table is already in 3NF because the column royaltyper depends on both columns of the primary key: the royalty percentage attributed to an author depends on the author and on the book. Conversely, the table Titles is in 2NF but not in 3NF because the city column may be determined both by the publisher name (pub_name) and by the primary key. The functional dependencies of the table show this straightforward situation:

\[
\begin{align*}
title_id & \rightarrow \text{pub\_name} \\
title_id & \rightarrow \text{city} \\
\text{pub\_name} & \rightarrow \text{city}
\end{align*}
\]

This relation table is nonetheless in 2NF because city is functionally dependent on the primary key. A table can be decomposed to achieve 3NF by doing the following:

1. Identify all the determinants amongst non-key columns and their dependent columns.
2. Create a new table from every determinant identified and their dependent columns. The determinant becomes the primary key of the new table.
3. Delete the dependent columns from the source table. Do not delete the determinant, since it will become the foreign key.

To achieve Third Normal Form in our example, we create a third table, called Publishers, containing pub_name and city, with pub_name becoming its primary key and deleting city from the Titles table (Figure 1.20).
FIGURE 1.20 Publishers and Titles tables in Third Normal Form

<table>
<thead>
<tr>
<th>title_id</th>
<th>pub_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU1032</td>
<td>Algodata Infosystems</td>
</tr>
<tr>
<td>BU1111</td>
<td>Algodata Infosystems</td>
</tr>
<tr>
<td>BU2075</td>
<td>New Moon Books</td>
</tr>
<tr>
<td>BU7832</td>
<td>Algodata Infosystems</td>
</tr>
<tr>
<td>PC1035</td>
<td>Algodata Infosystems</td>
</tr>
<tr>
<td>PC8888</td>
<td>Algodata Infosystems</td>
</tr>
<tr>
<td>PS3333</td>
<td>New Moon Books</td>
</tr>
<tr>
<td>TC7777</td>
<td>Binnet &amp; Hardley</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pub_name</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algodata Infosystems</td>
<td>Berkeley</td>
</tr>
<tr>
<td>Binnet &amp; Hardley</td>
<td>Washington</td>
</tr>
<tr>
<td>New Moon Books</td>
<td>Boston</td>
</tr>
</tbody>
</table>

Once in Third Normal Form, all the anomalies we encountered so far disappear:

- You can insert a new publisher even if it has not published a book.
- If you delete a royalty, you are not losing information about the publisher.
- The city of a publisher has to be updated in only one place.
- You may delete a row in the Titles table without simultaneously losing the information about the publisher.

The normalized logical model of our database is illustrated in Figure 1.21. It contains the three tables with the relationships and the keys.

FIGURE 1.21 The normalized logical model
3NF has many advantages. Amongst them, we find:

- Better data consistency.
- Data space is saved, because data occurs only once.
- Fewer anomalies.

In 99.99 percent of cases, 3NF is enough. Having achieved 3NF, you may have achieved higher normalization. Nevertheless, after E.F. Codd defined the first three normal forms, some gurus found issues in it. So, higher normal forms have been introduced. Let’s have a very quick look at these higher forms.

**Advanced Normalization**

The database community generally accepts three other levels of normal forms. These levels concern tables containing at least three columns that are all keys. These normal forms are the following:

**Boyce/Codd Normal Form**  Boyce/Codd Normal Form (BCNF) is a more precise version of the 3NF. It concerns a table that contains many composite overlapping candidate keys and is based on the concept of determinants. A relational table is in BCNF if and only if every determinant is a candidate key.

Review the definition of determinant and candidate key in the previous pages.

**Fourth Normal Form**  Fourth Normal Form (4NF) is based on the concept of multivalued dependency (MVD). A MVD can occur in a table containing at least three columns. If one column has multiple rows whose values are matching another column value of a single row, then there is a MVD. A table is in 4NF if it is in BCNF and if every MVD is also functionally dependent.
MVD is noted as ->>. A->>B means A multidetermines B. Given a table with three columns—A, B, and C—if a set of B values matching a pair of A and C values depends only on the A value and not on the C value, then A->>B.

**Fifth Normal Form**  
Fifth Normal Form (5NF) is based on the concept of join dependencies. Join dependency means that if a table is being decomposed into three or more tables, it can be joined again to retain its original state. A table is said to be in 5NF if it cannot be decomposed into smaller tables without the loss of data. In other words, if you add a row to a table that is not in 5NF, and if you decompose this table into smaller tables and join these tables again, the result you obtain contains spurious data.

If you are interested in going further than 5NF, I recommend that you read *An Introduction to Database Systems*, by Chris Date (Addison Wesley, 7th Edition, 1999). It’s a little bit academic, but one of the best books on database theory.

You may have thrown your book away after reading the definitions of these last normal forms. This is really complicated material. Lots of database specialists, if not all them, agree on the fact that most of the real-life tables in 3NF are also in 4NF and 5NF, so achieving the 3NF is the only requirement for a database. There may be less than a tenth of a percent of tables that need a real 4NF or 5NF analysis.

3NF guarantees that almost no redundancy remains in your database. But is it a good idea? While the situation is theoretically ideal, it may become unusable due to the number of tables and necessary joins to retrieve specific information. So, while we’re at it, let’s introduce redundancy into your 3NF database again!
The whole database community agrees on the 3NF requirement for a database. Nevertheless, if the result of the 3NF is the total or almost total elimination of data redundancy, it can lead to poor performance. Consider the relational model illustrated in Figure 1.22, directly extracted from the Northwind database.

If you want to calculate the total turnover realized with a specific customer, you must write a query that joins the three tables, calculate the amount of every order detail, and total all the amounts. That query will consume quite a lot of CPU time. Now consider adding the field Total-Amount to the table Orders. We obtain the relational model illustrated in Figure 1.23.
In Figure 1.23, the CompanyName column is required, which is why it is bolded. All the other columns allow the NULL value.

Now, when you want to calculate the total turnover realized with a specific customer, you just have to join two tables and calculate a sum. You could even add a field Total Turnover in the Customers table, if you need frequent access to this information. The global idea of denormalization is presented in this example: introducing redundancy to improve data access performance.

While denormalization has advantages, it also has drawbacks, the worst being the maintenance of redundant data. In the previous example, each time an order detail is inserted, the total amount of the order has to be calculated and updated in the order table, or in the customer table if you decided to store it with the customer’s data. Data integrity is endangered by denormalization, and update performance may decrease.

Data integrity is endangered because you have to guarantee that the redundant data are up to date. For example, you may decide that the Total Turnover column in the Customers table should be updated every night by a batch process recalculating every value, or that its value should
be calculated on the fly and cross-checked every night to correct possible inaccuracies. On the other hand, if you have to update the Customers table each time you insert a new order, you slow your insert query. Is the redundancy worth it?

Denormalization is a dangerous game and is generally more an art than a science. The techniques that are presented in this chapter give you an idea of what you can do with denormalization. Each time you denormalize your model, you must always thoroughly document your choice.

One last word before switching to the denormalization techniques: some database architects or consultants always denormalize a model or will advise you to do so, because they say that a model in 3NF cannot perform well. This is not necessarily true. Never predict performance problems before implementing the physical model because software and hardware have progressed, and what was true five or six years ago may not necessarily be true today. Also, every database is unique, and what is true for one system may be not be true for another; the volume of data, the number of users, the type of the server, of the network, the software used, and so on could be different. It creates a combination that has to be studied precisely before making any decision concerning de-normalization. Never denormalize before implementing your physical model and the first performance test is under full load.

We will cover the following denormalization techniques in the upcoming sections:

- Adding a redundant column
- Adding a derived column
- Partitioning tables

**Adding a Redundant Column**

Adding a redundant column is probably the most straightforward and logical denormalization technique. It consists of copying a column in a child table to a parent table. It generally violates the Third Normal Form, but it does help some queries to avoid a join. In the Pubs database, consider the Titles and Roysched tables (Figure 1.24).
The Roysched table contains the royalty range for each title. For example, if the sales of title BU1032 are between 0 and 5000, then the royalty is 10 percent, and above 5001 it is 12 percent. Now to avoid querying that table, the current value of royalty is inserted in the Titles table. Now, that table is not achieving 3NF anymore because the royalty column is functionally dependent on the title_id and ytd_sales columns. This column is not part of the primary key, so the table is not in 3NF anymore.

With the loss of the 3NF, anomalies can occur. Here are two examples:

- If a user updates the value of the royalty column in the Roysched table, he/she has to update the matching record in the Titles tables; otherwise, data is inconsistent.
- If a user updates the value of the ytd_sales column in the Titles table, he/she has to look for the corresponding royalty value in the Roysched tables to update the royalty column.

To avoid these two situations, it is possible to create an update trigger on each table to track updates of the royalty column of the Roysched table and of the ytd_sales column of the Titles table. The trigger is a piece of code fired during the update of one of the columns. Compared to a single update, the trigger slows the overall update. That loss of performance may be a minor drawback compared to the fact that each time a title is queried, the user retrieves its royalty value without having to query another table or to join that table.
Adding a Derived Column

Another useful technique of denormalization is the use of derived columns. A derived column is a column whose values are calculated from the values of one or many other columns of the same table or other tables. Adding such a column generally violates the 3NF, since this column is functionally dependent on non-key columns.

The simplest example is the computed column: In a Sales table, you store the amount, the sales tax, and the net price, calculated from the amount and the sales tax.

The Titles and Sales table in Figure 1.25 illustrates a more complicated example.

**FIGURE 1.25** Titles and Sales table

![Diagram of Titles and Sales table]

Each time you wish to know the year-to-date sales of a given book, you need to query the Sales table and total the values of the qty column for that book. It may be a long-running query if the sales table is big. To avoid querying that table and totaling the values, the architect introduced the ytd_sales column in the Titles table. Now each time you query the sales of a given book, you just have to query the Titles column. Of course, as for
the redundant column, the value of that column needs to be maintained dynamically to be consistent and accurate.

You can add triggers to the Sales table to update the ytd_sales column of the Titles table each time a sales record is inserted, deleted, or updated. This trigger will lower the performance and inserts, deletes, and updates. But again, the performance gain of the data retrieval must outweigh the performance loss of the insert, delete, and update operations.

**Partitioning Tables**

Partitioning a table is not really a denormalizing technique, but it is worth mentioning because it can address particular performance issues. There are two ways to partition a table: horizontally or vertically.

**Vertical Partitioning**

Vertical partitioning consists of cutting the table in two or more tables by moving entire columns. Consider the example illustrated in Figure 1.26.

**Figure 1.26** Vertical partitioning

![Vertical partitioning diagram](image)

The Publishers table has been split into two tables. One (Publishers) contains all the “basic” information, and the other (Pub_info) contains the logo and the pr_info field. This split has been realized for two reasons:

- There is not a logo and a description for every publisher, so it makes more sense to split mandatory information from optional information.
- The fields in the Pub_info table are large binary objects (BLOB), and the architect may want to store them in another disk or “table space.”
SQL Server 2000 allows you to store text and image columns on another filegroup thanks to the clause TEXTIMAGE_ON of the CREATE TABLE statement. See Chapter 2: Database Physical Modeling, for more information on the CREATE TABLE statement.

Another interesting point concerning vertical partitioning is the table width and the number of records per page. In SQL Server 2000, an 8K page contains a certain number of records. The wider the table, the fewer the records per page. The cache hit ratio may increase, the number of I/O per operation may lower, and the SQL Server cache may be well used.

In splitting a table for performance purposes, you should consider keeping the columns that are accessed more frequently in the “master” table and moving the other columns to one or more “slave” tables. Then, a one-to-one relationship between the master and each slave table guarantees the referential integrity.

**Horizontal Partitioning**

Another classic way of partitioning a table consists of moving a certain number of rows to one or many other tables. This is done during archiving, for example. If you consider a Sales table, you can imagine that every July the sales from July of last year to June of this year are archived. This technique is fine to keep small tables for the transactional system, while still allowing access to the archived data.

A view can be used to simulate a full view of archived and live data. With the new feature of partitioned view of SQL Server 2000, this technique becomes very interesting to achieve scaling out.

Other examples can be found in real-world applications, like splitting customers from prospects, active customers from customers who have not placed an order for more than 12 months, and so on.
Summary

This chapter is the only entirely theoretical one of the book. It may be hard to remember all the terms and concepts we have learned here. But it's the kind of information you will use all your database life long, because you cannot create a good database application without keeping these concepts in mind.

In this chapter, we covered the following:

- Designing a database system
- The Entity/Relationship model
- The relational model and the normalization process
- The denormalization process

Exam Essentials

Know what makes a good database design. In the exam, you will be judged on your real-world knowledge. Knowing what makes a good database design will enable you to focus on the technical questions and tricks.

Identify entities and attributes. The basis of ER modeling is the identification of entities and attributes. Having a thorough knowledge of modeling will help you criticize the way a database is designed and will help you to create a good design.

Identify the types of relationships. Even if one-to-one or one-to-many relationships are obvious, you should know how to manage every type of relationship, even many-to-many.

Know how to define key attributes. Candidate keys, primary keys, and alternate keys are the identification keys of your entities. Foreign keys are the basis of relationships. Defining them will allow you to enforce entity and referential integrity.
Identify precisely all the integrity types. Integrity is the source of correct data. Know the four types of integrity, what they are used for, and how they can be enforced to design a precise and optimal ER model.

Know how to normalize and denormalize an ER model. You should have no problems with normal forms, at least up to the third. Denormalization techniques are commonly used and can appear in the exam.

Key Terms

Before you take the exam, be certain you are familiar with the following terms:

- alternate keys
- artificial key
- attribute
- binary attributes
- business rules
- candidate key
- cardinality
- conceptual design
- denormalization
- derived column
- determinant
- direction
- domain
- domain integrity
- enterprise integrity
- entity
- entity integrity
- ER model
- existence
- fixed point
- floating point
- foreign key
- full functional dependency
- character attributes
- Integrity
- key attribute
- logical design
- logical model
- many-to-many relationship
- non-key attribute
- normal form
- Normal form theory
- normalization process
- Numeric attributes
- one-to-many relationship
- one-to-one relationship
- physical model
- primary key
- referential integrity
- Relational Database Management Systems (RDBMS)
- relational model
- relational table
- relationship
- special attributes
- transitive dependency
1. You are a developer for World Wide Importers. You are designing the new shipment tracking system. You print your ER model to show some selected users during the next phasing meeting (see graphic).
What should you add to your model to ensure a useful meeting with your end-users?

A. Attributes  
B. Entities  
C. Datatypes  
D. Relationships

2. Build a list: As a database consultant, you have been asked to optimize a database model designed by the IT department of an insurance company. The model comprises just entities and attributes. You should follow a certain number of steps before producing an optimized ready-to-implement model. What is the proper sequence of steps to produce this model? Some elements may not be part of the sequence.

<table>
<thead>
<tr>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define primary keys</td>
</tr>
<tr>
<td>Denormalize the model</td>
</tr>
<tr>
<td>Define attributes</td>
</tr>
<tr>
<td>Normalize the model</td>
</tr>
<tr>
<td>Define alternate keys</td>
</tr>
<tr>
<td>Define entities</td>
</tr>
<tr>
<td>Define relationships</td>
</tr>
</tbody>
</table>

3. You are a developer for World Wide Importers. One developer of your team is working on the Products entity. Each product is imported by one supplier only. You need to record information on every product and on every supplier. Your developer shows you the structure of the table he designed and a data sample (see graphics on next page).
As a matter of fact, you discover that the table needs a little extra work. In which normal form is it?

A. First
B. Second
C. Third
D. Boyce-Codd

4. You are developing a new customer care system for an insurance company. Every customer will be assigned a unique customer ID made of a combination of 7 characters and 8 figures. You expect to have over one million customers, each signing an average of 2.5 policies. Furthermore, it is important to track every customer’s questions and complaints. You expect over 10 questions and complaints per customer. You want to minimize space used in your database. What primary key are you going to define for the customers table to minimize space and programming tasks?

A. An integer column, defined with an auto-numbering property
B. The customer ID
C. A unique identifier column, designed to generate a new globally unique ID for every row

D. A big integer column, defined with an auto-numbering property

5. You are a database developer for a banking corporation. Recently, one of the counter clerks “lost” a customer. This customer went to the bank for a deposit, but the counter clerk could not find her by her customer ID. In fact, after a few minutes’ search, he found her, but with a wrong ID. After looking at the audit tables, it seemed another counter clerk accidentally modified the customer ID. You need to forbid the modification of the primary key to avoid any other “loss” like this. In SQL Server 2000, what is the fastest way to implement this feature, without modifying the existing front-end application?

A. Alter the table to enable the CHECK PRIMARY KEY option.

B. Alter the table to disable the MODIFY PRIMARY KEY option.

C. Add an AFTER UPDATE trigger to the table that rolls back the transaction in case of modification of the primary key value.

D. It is not possible to implement this feature in SQL Server.

6. The database application you developed last year for the insurance company you are working with was performing well, until last month when some users started to complain about some long-running queries. Last month your company acquired another insurance company and inserted all its existing customers and policies into the database, increasing the volume of data by a magnitude of 3. You have been asked to find the cheapest solution to this performance problem before the end of the week. After having analyzed what was happening, you have observed that only 10 percent of the data is used 90 percent of the time. In fact, data older than 2 years are selected only in 0.5 percent of the time. What is the solution you are going to implement?

A. Buy a new RAID 5 subsystem and spread the data all over the disks.
B. Change the server to a new 4-way machine.

C. It is not possible to enhance performance before the end of the week. You need more time.

D. Split the data horizontally and store the archive table on another disk.

7. The database application you developed last year for the regional bank you are working with was performing well, until last month when some users started to complain about some long-running queries. Last month your company acquired another bank and inserted all its existing customers and accounts into the database, increasing the volume of data by a magnitude of 5. You have been asked to find the cheapest solution to this performance problem before the end of the week. After having analyzed what was happening, you have observed that each time an employee was gaining access to a customer record, the system was calculating the amount of money of his account based on all the money transferred since the beginning of the year. What is the solution you are going to implement to hasten the access to the customer record?

A. Create a stored procedure that calculates the amount on the fly.

B. Denormalize the customer table to include the calculated amount value, updated through a batch that runs every night.

C. Index the transfer table to fasten the join with the customers table.

D. Create a temporary table in tempdb that stores the account’s amount and query this table each time a customer is queried.

8. You are helping your town library develop their new computer system to track members, books, and borrowed books. The manager of the library is a computer addict but knows little of ER modeling. She designed a Member table to store every member, assigning each a unique ID. She designed a Book table to store every book of the library, and a Borrowed table to assign every borrowed book to members. Each member can only borrow three books at a time, can
keep them up to four weeks and must bring them back at the same
time. The proposed design is illustrated in the following graphic:

What level of normal form is reached by this model?

A. None
B. 1NF
C. 2NF
D. 3NF

9. You are a SQL Server developer for Northwind Traders. Your users
complain about performance of the application when they query the
order amount per employee name and per customer name. After a
quick investigation, you discover that this is due to the number of
tables joined to calculate the amount. Your logical design is
represented in the graphic on the next page.
What can you do to optimize this query?

A. Create a new denormalized table containing the employee name, the customer name, and the amount ordered, and create the necessary trigger to maintain this table.

B. Create a stored procedure that performs the needed calculation.

C. Create new indexes on the Order Details table.

D. Create a view on the four tables.

10. You are a database developer for the local university. You need to define a relationship between the students and the courses, knowing a student can attend many courses and one course can be attended by many students. You have the two entities illustrated in the following graphic:
How can you implement this many-to-many relationship?

A. Insert a StudentID attribute in the Courses table and a CourseID column in the Students table.

B. Create a new entity called StudentsCourses containing at least two attributes, StudentID and CourseID, forming the primary key.

C. Insert a StudentID attribute in the Course table.

D. Insert a CourseID attribute in the Students table.

11. One of the developers on your team asked you about a problem he could not address. He needs to represent a hierarchy in the new HR database. Every employee reports to a manager. Managers can report to another manager and so on, up to the CEO. He explains to you that there are five hierarchical levels in the company, so he intends to create five entities representing every level. You think this can lead to many problems, the first being the case of a promotion. A promoted employee has to be moved from one level to an upper level, and that could lead to consistency issues. What is the best solution, using the ER model, to address such a hierarchy?

A. Use two tables, one containing the employee information and one the hierarchy information.

B. It is impossible to address this problem in a relational database.

C. Keep a table for every hierarchical level and develop a series of stored procedures to manage inserts, deletes, and updates.

D. Insert a ReportsTo column in the Employees table and create a recursive relationship.

12. As an independent SQL Server expert, you have been chosen to explain the ER modeling that the developers will use to model the needed data and business processes to users of the future loyalty management system. You decide to define the basic objects of the ER model. What are they? (Chose three.)

A. Entity

B. Relationship

C. Datatype
13. You are a SQL Server database developer for Northwind Traders. You designed the Products, Categories, and Suppliers table illustrated in the graphic below.

What is the level of normalization of your model?

A. 1NF
B. 2NF
C. 3NF
D. BCNF

14. You are designing a new procurement database for a regional bank. While defining the suppliers and orders relationship, you are faced with the choice of what has to be done when a supplier is deleted from the database. You must propose all the SQL Server possible declarative choices to your customer. What are they? (Choose all that apply.)

A. Restrict: You cannot delete a supplier if it is linked with existing orders.

B. No Effect: You can delete a supplier even if it is linked with existing orders.
C. Default: All foreign keys are defined to a default value if the matching primary key is deleted.

D. Cascade: Every order is deleted if the linked supplier is deleted.

15. You are a database developer for an insurance company. Every insurance policy is managed by one and only one product manager. One product manager can manage many policies. How can you represent that information in the database?

A. Include the manager ID in the Policies entity.

B. Include the policy ID in the Managers entity.

C. Create a relationship entity formed by the policy ID and the manager ID.

D. Include the manager ID in the Policies entity and the policy ID in the Managers entity.

Answers to Review Questions

1. D. Attributes and entities are represented in the diagram. End-users do not need to be know about datatypes, but they do need to know about relationships to fully understand the links between entities.

2. Define primary keys
   Normalize the model
   Define relationships
   Denormalize the model

Defining primary keys first allows you to enforce entity integrity and prepare the normalization process that is based on functional dependency to the primary key. Normalizing the database leads naturally to the definition of relationships. You will then have a 3NF model that you will only need to denormalize if needed.
3. B. Some non-key attributes depend on other non-key attributes. For example, Country depends on the supplier name and product ID. This is a transitive dependency: if you have the product ID, you can find the supplier name, and once you get the supplier name, you can find the supplier country. So, if you know the product ID, you know the supplier country. Country is not functionally dependent only on the primary key, but also on the Company Name.

4. A. For one million rows, the integer is sufficient (it goes up to more than 2 billion) and will consume only 4 bytes per row instead of 15 for the customer ID, 16 for the unique identifier, and 8 for the big integer.

5. C. Options A and B do not exist. C is the only way to do it without modifying the programming logic.

6. D. The 3NF seems to be the problem in the sense that there is too much data in the table and the machine is probably not suited for that volume of data. So, the cheapest solution is to split the table horizontally and store older data on another disk to minimize the volume of data in memory.

7. B. This is a classic problem of heavy calculation on a frequently accessed table. The only solution is to denormalize the table with a column updated through a batch or a trigger, depending on the frequency of inserts and deletes.

8. A. The Borrowing table contains a repeating group. To be in First Normal Form, the table should only contain one BookID column, not three.

9. A. Creating a new table will give the best results since the information will be immediately available. However, the information should be updated through synchronous or asynchronous mechanisms.
10. B. The only way to implement a many-to-many relationship in ER modeling is by creating a new entity made of, at least, the primary keys of the linked entities.

11. D. This is the classical hierarchy problem. In such a case, the only solution is a recursive relationship, which handles the hierarchy.

12. A, B, and D. ER stands for Entity-Relationship, and an entity is made of attributes.

13. B. This is a tricky question. At first sight, the model is in 3NF, but there is a transitive dependency in the Suppliers table: ContactTitle depends on ContactName and not only on SupplierID. You should introduce a Contacts table to be in 3NF.

14. A, B, and D. This is a tricky one. Options A and D are obvious, but B is possible if you do not enforce the foreign key in the Orders table. C is only possible through stored procedures or triggers.

15. A. This is a one-to-many relationship. Each policy can have only one manager ID, so the manager ID must be part of the Policies table.