CHAPTER 13

Designing Your Database

About this chapter

This chapter introduces the basic concepts of relational database design and gives you step-by-step suggestions for designing your own databases. It uses the expedient technique known as *conceptual data modeling*, which focuses on entities and the relationships between them.

Contents

Introduction

While designing a database is not a difficult task for small and medium sized databases, it is an important one. Bad database design can lead to an inefficient and possibly unreliable database system. Because client applications are built to work on specific parts of a database, and rely on the database design, a bad design can be difficult to revise at a later date.

☞ This chapter covers database design in an elementary manner. For more advanced information, you may wish to the DataArchitect documentation. DataArchitect is a component of Powersoft PowerDesigner, a database design tool available from Sybase, Inc.

☞ You may also wish to consult an introductory book such as *A Database Primer* by C. J. Date. If you are interested in pursuing database theory, C. J. Date's *An Introduction to Database Systems* is an excellent textbook on the subject.

Java classes and database design

The addition of Java classes to the available data types extends the relational database concepts on which this chapter is based. Database design involving Java classes is not discussed in this chapter.

☞ For information on designing databases that take advantage of Java class data types, see "Java database design" on page 495.

Database design concepts

In designing a database, you plan what things you want to store information about, and what information you will keep about each one. You also determine how these things are related. In the common language of database design, what you are creating during this step is a **conceptual database model**.

Entities and relationships

The distinguishable objects or *things* that you want to store information about are called **entities**. The *associations* between them are called **relationships**. You might like to think of the entities as nouns in the language of database description and the relationships as verbs.

Conceptual models are useful because they make a clean distinction between the entities and relationships. These models hide the details involved in implementing a design in any particular database management system. They allow you to focus on fundamental database structure. Hence, they also form a common language for the discussion of database design.

Entity-relationship diagrams

The main component of a conceptual database model is a diagram that shows the entities and relationships. This diagram is commonly called an **entity-relationship diagram**. In consequence, many people use the name entity-relationship modeling to refer to the task of creating a conceptual database model.

Conceptual database design is a top-down design method. There are now sophisticated tools such as Powersoft PowerDesigner that helps you pursue this method, or other approaches. This chapter is an introductory chapter only, but it does contain enough information for the design of straightforward databases.

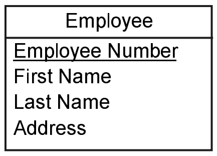
Entities

An entity is the database equivalent of a noun. Distinguishable objects such as employees, order items, departments and products are all examples of entities. In a database, a table represents each entity. The entities that you build into your database arise from the activities for which you will be using the database, whether that be tracking sales calls, maintaining employee information, or some other activity.

Attributes and identifiers

Each entity contains a number of **attributes**. Attributes are particular characteristics of the things that you would like to store. For example, in an employee entity, you might want to store an employee ID number, first and last names, an address, and other particular information that pertains to a particular employee. Attributes are also known as *properties*.

You depict an entity using a rectangular box. Inside, you list the attributes associated with than entity.



An **identifier** is one or more attributes on which all the other attributes depend. It uniquely identifies an item in the entity. Underline the names of attributes that you wish to form part of an identifier.

In the Employee entity, above, the Employee Number uniquely identifies an employee. All the other attributes store information that pertains only to that one employee. For example, an employee number uniquely determines an employee's name and address. Two employees might have the same name or the same address, but you can make sure that they don't have the same employee number. Employee Number is underlined to show that it is an identifier.

It is good practice to create an identifier for each entity. As will be explained later, these identifiers become primary keys within your tables. Primary key values must be unique and cannot be null or undefined. They identify each row in a table uniquely and improve the performance of the database server.

Relationships

A relationship between entities is the database equivalent of a verb. An employee is a member of a department, or an office is located in a city. Relationships in a database may appear as foreign key relationships between tables, or may appear as separate tables themselves. You will see examples of each in this chapter.

The relationships in the database are an encoding of rules or practices that govern the data in the entities. If each department has one department head, you can create a one-to-one relationship between departments and employees to identify the department head.

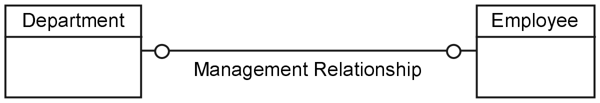
Once a relationship is built into the structure of the database, there is no provision for exceptions. There is nowhere to put a second department head. Duplicating the department entry would involve duplicating the department ID, which is the identifier. Duplicate identifiers are not allowed.

Tip
Strict database structure can benefit you, because it can eliminate inconsistencies, such as a department with two managers. On the other hand, you as the designer should make your design flexible enough to allow some expansion for unforeseen uses. Extending a well-designed database is usually not too difficult, but modifying the existing table structure can render an entire database and its client applications obsolete.

Cardinality of relationships

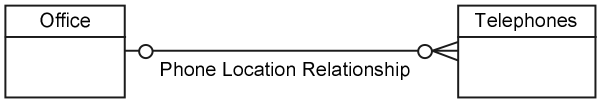
There are three kinds of relationships between tables. These correspond to the **cardinality** (number) of the entities involved in the relationship.

♦ **One-to-one relationships** You depict a relationship by drawing a line between two entities. The line may have other markings on it such as the two little circles shown. Later sections explain the purpose of these marks.



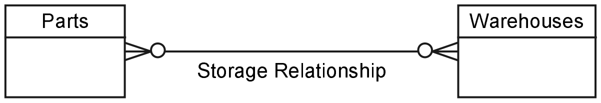
One employee manages *one* department.

♦ **One-to-many relationships** The fact that one item contained in Entity 1 can be associated with multiple entities in Entity 2 is denoted by the multiple lines forming the attachment to Entity 2.



One office can have *many* telephones.

♦ **Many-to-many relationships** In this case, draw multiple lines for the connections to both entities.



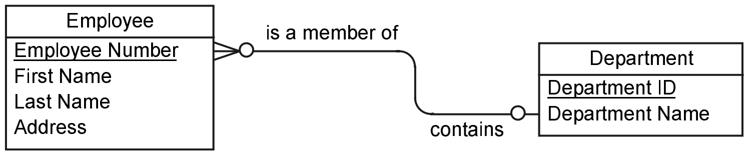
One warehouse can hold *many* different parts and one type of part can be stored at *many* warehouses.

Roles

You can describe each relationship with two **roles**. Roles are verbs or phrases that describe the relationship from each point of view. For example, a relationship between employees and departments might be described by the following two roles.

1 An employee *is a member of* a department.

2 A department *contains* an employee.



Roles are very important because they afford you a convenient and effective means of verifying your work.

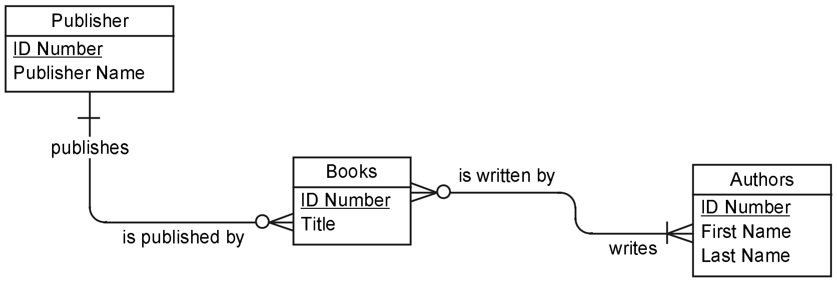
Tip
Whether reading from left-to-right or from right-to-left, the following rule makes it easy to read these diagrams: Read the

1. name of the first entity,
2. role next to the *first entity*,
3. cardinality from the connection to the *second entity*, and
4. name of the second entity.

Mandatory elements

The little circles just before the end of the line that denotes the relation serve an important purpose. A circle means that an element can exist in the one entity without a corresponding element in the other entity.

If a cross bar appears in place of the circle, that entity must contain *at least* one element for each element in the other entity. An example will clarify these statements.



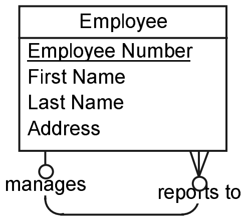
This diagram corresponds to the following four statements.

1. A publisher publishes *zero or more* books.
2. A book is published by *exactly one* publisher.
3. A book is written by *one or more* authors.
4. An author writes *zero or more* books.

Tip
Think of the little circle as the digit 0 and the cross bar as the number one. The circle means *at least zero*. The cross bar means *at least one*.

Reflexive relationships

Sometimes, a relationship will exist between entries in a single entity. In this case, the relationship is called **reflexive**. Both ends of the relationship attach to a single entity.



This diagram corresponds to the following two statements.

1 An employee reports to at most one other employee.

2 An employee manages zero or more or more employees.

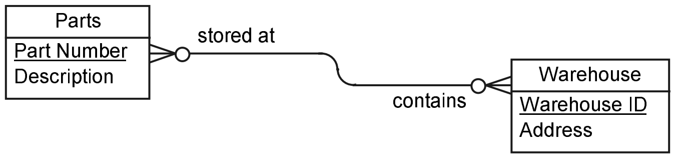
Notice that in the case of this relation, it is essential that the relation be optional in both directions. Some employees are not managers. Similarly, at least one employee should head the organization and hence report to no one.

Naturally, you would also like to specify that an employee cannot be his or her own manager. This restriction is a type of *business rule*. Business rules are discussed later as part of "The design process" on page 331.

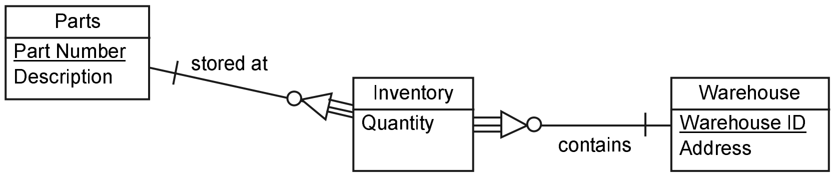
Changing many-to-many relationships into entities

When you have attributes associated with a *relationship*, rather than an entity, you can change the relationship into an entity. This situation sometimes arises with many-to-many relationships, when you have attributes that are particular to the relationship and so you cannot reasonably add them to either entity.

Suppose that your parts inventory is located at a number of different warehouses. You have drawn the following diagram.



But you wish to record the quantity of each part stored at each location. This attribute can only be associated with the relationship. Each quantity depends on both the parts and the warehouse involved. To represent this situation, you can redraw the diagram as follows:



Notice the following details of the transformation:

1 Two new relations join the relation entity with each of the two original entities. They inherit their names from the two roles of the original relationship: *stored at* and *contains*, respectively.

2 Each entry in the Inventory entity demands one mandatory entry in the Parts entity and one mandatory entry in the Warehouse entity. These relationships are mandatory because a storage relationship only makes sense if it is associated with one particular part and one particular warehouse.

3 The new entity is dependent on both the Parts entity and on the Warehouse entity, meaning that the new entity is identified by the identifiers of both of these entities. In this new diagram, one identifier from the Parts entity and one identifier from the Warehouse entity uniquely identify an entry in the Inventory entity. The triangles that appear between the circles and the multiple lines that join the two new relationships to the new Inventory entity denote the dependencies.

Do not add either a Part Number or Warehouse ID attribute to the Inventory entity. Each entry in the Inventory entity does depend on both a particular part and a particular warehouse, but the triangles denote this dependence more clearly.

The design process

There are five major steps in the design process.

- "Step 1: Identify entities and relationships" on page 331.
- "Step 2: Identify the required data" on page 334.
- "Step 3: Normalize the data" on page 336.
- "Step 4: Resolve the relationships" on page 339.
- "Step 5: Verify the design" on page 342.

For information about implementing the database design, see "Working with Database Objects" on page 65.

Step 1: Identify entities and relationships

❖ **To identify the entities in your design and their relationship to each other:**

1 **Define high-level activities** Identify the general activities for which you will use this database. For example, you may want to keep track of information about employees.

2 **Identify entities** For the list of activities, identify the subject areas you need to maintain information about. These subjects will become entities. For example, hire *employees*, assign to a *department*, and determine a *skill* level.

3 **Identify relationships** Look at the activities and determine what the relationships will be between the entities. For example, there is a relationship between parts and warehouses. Define two roles to describe each relationship.

4 **Break down the activities** You started out with high-level activities. Now, examine these activities more carefully to see if some of them can be broken down into lower-level activities. For example, a high-level activity such *as maintain employee information* can be broken down into:

   - Add new employees.
   - Change existing employee information.
   - Delete terminated employees.

5. **Identify business rules** Look at your business description and see what rules you follow. For example, one business rule might be that a department has one and only one department head. These rules will be built into the structure of the database.

Entity and relationship example

Example

ACME Corporation is a small company with offices in five locations. Currently, 75 employees work for ACME. The company is preparing for rapid growth and has identified nine departments, each with its own department head.

To help in its search for new employees, the personnel department has identified 68 skills that it believes the company will need in its future employee base. When an employee is hired, the employee's level of expertise for each skill is identified.

Define high-level activities

Some of the high-level activities for ACME Corporation are:

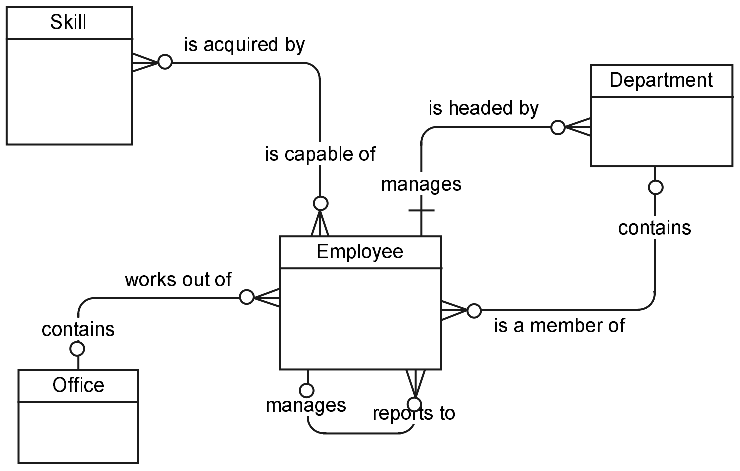
- Hire employees.
- Terminate employees.
- Maintain personal employee information.
- Maintain information on skills required for the company.
- Maintain information on which employees have which skills.
- Maintain information on departments.
- Maintain information on offices.

Identify the entities and relationships

Identify the entities (subjects) and the relationships (roles) that connect them. Create a diagram based on the description and high-level activities.

Use boxes to show entities and lines to show relationships. Use the two roles to label each relationship. You should also identify those relationships that are one-to-many, one-to-one, and many-to-many using the appropriate annotation.

Below, is a rough entity-relationship diagram. It will be refined throughout the chapter.



Break down the high-level activities

The following lower-level activities below are based on the high-level activities listed above:

- ♦ Add or delete an employee.
- ♦ Add or delete an office.
- ♦ List employees for a department.
- ♦ Add a skill to the skill list.
- ♦ Identify the skills of an employee.
- ♦ Identify an employee's skill level for each skill.
- ♦ Identify all employees that have the same skill level for a particular skill.
- ♦ Change an employee's skill level.

These lower-level activities can be used to identify if any new tables or relationships are needed.

Identify business rules

Business rules often identify one-to-many, one-to-one, and many-to-many relationships.

The kind of business rules that may be relevant include the following:

- ♦ There are now five offices; expansion plans allow for a maximum of ten.
- ♦ Employees can change department or office.
- ♦ Each department has one department head.
- ♦ Each office has a maximum of three telephone numbers.
- ♦ Each telephone number has one or more extensions.

- When an employee is hired, the level of expertise in each of several skills is identified.
- Each employee can have from three to twenty skills.
- An employee may or may not be assigned to an office.

Step 2: Identify the required data

❖ **To identify the required data:**

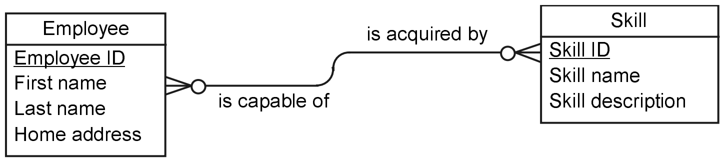
1. Identify supporting data.
2. List all the data you need to track.
3. Set up data for each entity.
4. List the available data for each entity. The data that describes an entity (subject) answers the questions who, what, where, when, and why.
5. List any data required for each relationship (verb).
6. List the data, if any, that applies to each relationship.

Identify supporting data

The supporting data you identify will become the names of the attributes of the entity. For example, the data below might apply to the Employee entity, the Skill entity, and the Expert In relationship.

Employee	Skill	Expert In
Employee ID	Skill ID	Skill level
Employee first name	Skill name	Date skill was acquired
Employee last name	Description of skill	
Employee department		
Employee office		
Employee address		

If you make a diagram of this data, it will look something like this picture:



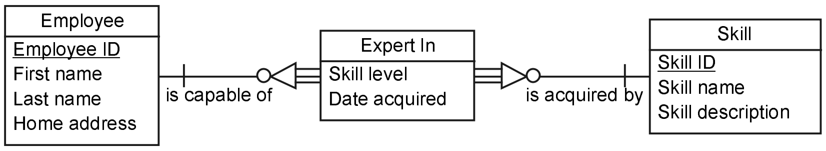
Observe that not all of the attributes you listed appear in this diagram. The missing items fall into two categories:

1 Some are contained implicitly in other relationships; for example, Employee department and Employee office are denoted by the relations to the Department and Office entities, respectively.

2 Others are not present because they are associated not with either of these entities, but rather the relationship between them. The above diagram is inadequate.

The first category of items will fall naturally into place when you draw the entire entity-relationship diagram.

You can add the second category by converting this many-to-many relationship into an entity.



The new entity depends on both the Employee and the Skill entities. It borrows its identifiers from these entities because it depends on both of them.

Things to remember

♦ When you are identifying the supporting data, be sure to refer to the activities you identified earlier to see how you will access the data.

For example, you may need to list employees by first name in some situations and by last name in others. To accommodate this requirement, create a First Name attribute and a Last Name attribute, rather than a single attribute that contains both names. With the names separate, you can later create two indexes, one suited to each task.

♦ Choose consistent names. Consistency makes it easier to maintain your database and easier to read reports and output windows.

For example, if you choose to use an abbreviated name such as Emp_status for one attribute, you should not use a full name, such as Employee_ID, for another attribute. Instead, the names should be Emp_status and Emp_ID.

♦ At this stage, it is not crucial that the data be associated with the correct entity. You can use your intuition. In the next section, you'll apply tests to check your judgment.

Step 3: Normalize the data

Normalization is a series of tests that eliminate redundancy in the data and make sure the data is associated with the correct entity or relationship. There are five tests. This section presents the first three of them. These three tests are the most important and so the most frequently used.

Why normalize?
The goals of normalization are to remove redundancy and to improve consistency. For example, if you store a customer's address in multiple locations, it is difficult to update all copies correctly should he move.

For more information about the normalization tests, see a book on database design.

Normal forms

There are several tests for data normalization. When your data passes the first test, it is considered to be in first normal form. When it passes the second test, it is in second normal form, and when it passes the third test, it is in third normal form.

❖ **To normalize data in a database:**

1 List the data.

- ♦ Identify at least one key for each entity. Each entity must have an identifier.
- ♦ Identify keys for relationships. The keys for a relationship are the keys from the two entities that it joins.
- ♦ Check for calculated data in your supporting data list. Calculated data is not normally stored in a relational database.

2 Put data in first normal form.

- ♦ If an attribute can have several different values for the same entry, remove these repeated values.
- ♦ Create one or more entities or relationships with the data that you remove.

3 Put data in second normal form.

- ♦ Identify entities and relationships with more than one key.
- ♦ Remove data that depends on only one part of the key.
- ♦ Create one or more entities and relationships with the data that you remove.

4 Put data in third normal form.

- ♦ Remove data that depends on other data in the entity or relationship, not on the key.
- ♦ Create one or more entities and relationships with the data that you remove.

Data and identifiers

Before you begin to normalize (test your design), simply list the data and identify a unique identifier each table. The identifier can be made up of one piece of data (attribute) or several (a compound identifier).

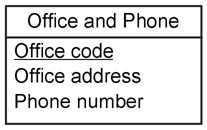
The identifier is the set of attributes that uniquely identifies each row in an entity. The identifier for the Employee entity is the Employee ID attribute. The identifier for the Works In relationship consists of the Office Code and Employee ID attributes. You can make an identifier for each relationship in your database by taking the identifiers from each of the entities that it connects. In the following table, the attributes identified with an asterisk are the identifiers for the entity or relationship.

Entity or *Relationship*	**Attributes**
Office	*Office code Office address Phone number
Works in	*Office code *Employee ID
Department	*Department ID Department name
Heads	*Department ID *Employee ID
Member of	*Department ID *Employee ID
Skill	*Skill ID Skill name Skill description
Expert in	*Skill ID *Employee ID Skill level Date acquired
Employee	*Employee ID last name first name Social security number Address phone number date of birth

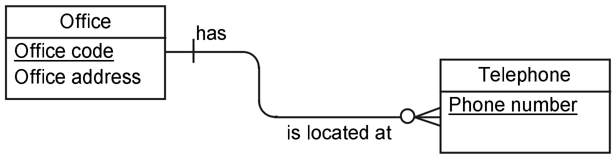
Putting data in first normal form

- To test for first normal form, look for attributes that can have repeating values.
- Remove attributes when multiple values can apply to a single item. Move these repeating attributes to a new entity.

In the entity below, Phone number can repeat—an office can have more than one telephone number.



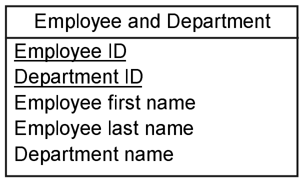
Remove the repeating attribute and make a new entity called Telephone. Set up a relationship between Office and Telephone.



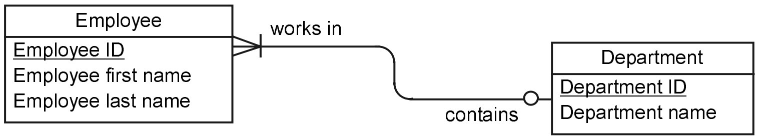
Putting data in second normal form

- Remove data that does not depend on the whole key.
- Look only at entities and relationships whose identifier is composed of more than one attribute. To test for second normal form, remove any data that does not depend on the whole identifier. Each attribute should depend on all of the attributes that comprise the identifier.

In this example, the identifier of the Employee and Department entity is composed of two attributes. Some of the data does not depend on both identifier attributes; for example, the department name depends on only one of those attributes, Department ID, and Employee first name depends only on Employee ID.



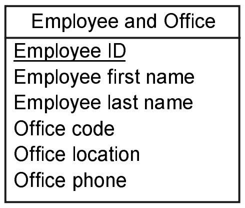
Move the identifier Department ID, which the other employee data does not depend on, to a entity of its own called Department. Also move any attributes that depend on it. Create a relationship between Employee and Department.



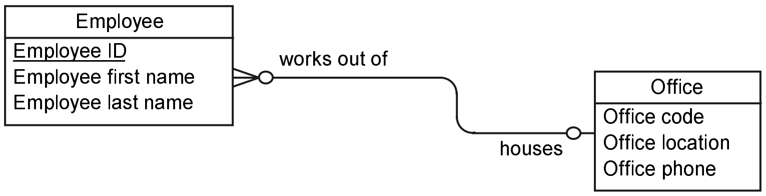
Putting data in third normal form

- Remove data that doesn't depend directly on the key.
- To test for third normal form, remove any attributes that depends on other attributes, rather than directly on the identifier.

In this example, the Employee and Office entity contains some attributes that depend on its identifier, Employee ID. However, attributes such as Office location and Office phone depend on another attribute, Office code. They do not depend directly on the identifier, Employee ID.



Remove Office code and those attributes that depend on it. Make another entity called Office. Then, create a relationship that connects Employee with Office.



Step 4: Resolve the relationships

When you finish the normalization process, your design is almost complete. All you need to do is to *generate* the **physical data model** that corresponds to your conceptual data model. This process is also known as resolving the relationships, because a large portion of the task involves converting the relationships in the conceptual model into the corresponding tables and foreign-key relationships.

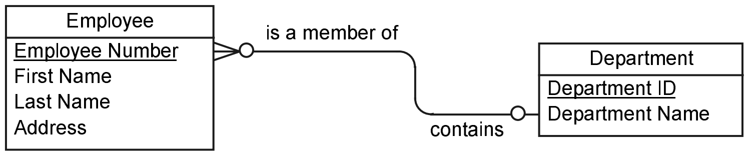
Whereas the conceptual model is largely independent of implementation details, the physical data model is tightly bound to the table structure and options available in a particular database application. In this case, that application is Adaptive Server Anywhere.

Resolving relationships that do not carry data

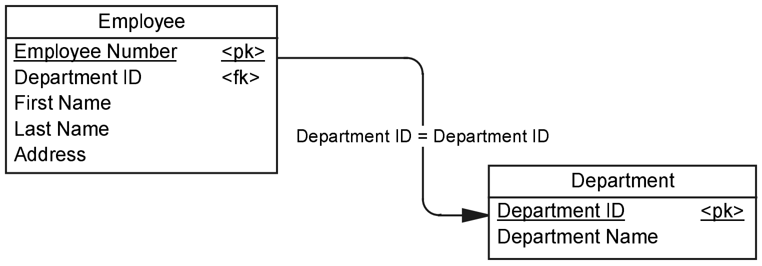
In order to implement relationships that do not carry data, you define foreign keys. A **foreign key** is a column or set of columns that contains primary key values from another table. The foreign key allows you to access data from more than one table at one time.

A database design tool such as the DataArchitect component of Powersoft PowerDesigner can generate the physical data model for you. However, if you're doing it yourself there are some basic rules that help you decide where to put the keys.

- **One to many** An one-to-many relationship always becomes an entity and a foreign key relationship.

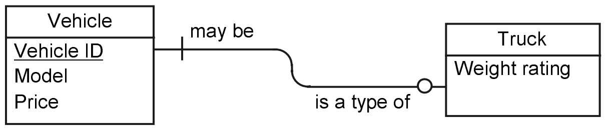


Notice that entities become tables. Identifiers in entities become (at least part of) the primary key in a table. Attributes become columns. In a one-to-many relationship, the identifier in the *one* entity will appear as a new foreign key column in the *many* table.

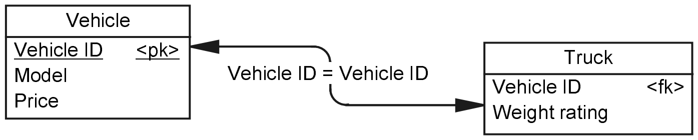


In this example, the Employee *entity* becomes an Employee *table*. Similarly, the Department entity becomes a Department table. A foreign key called Department ID appears in the Employee table.

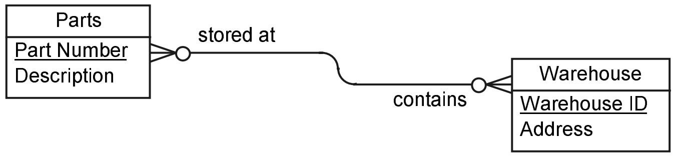
- **One to one** In a one-to-one relationship, the foreign key can go into either table. If the relationship is mandatory on one side, but optional on the other, it should go on the optional side. In this example, put the foreign key (Vehicle ID) in the Truck table because a vehicle does not have to be a truck.



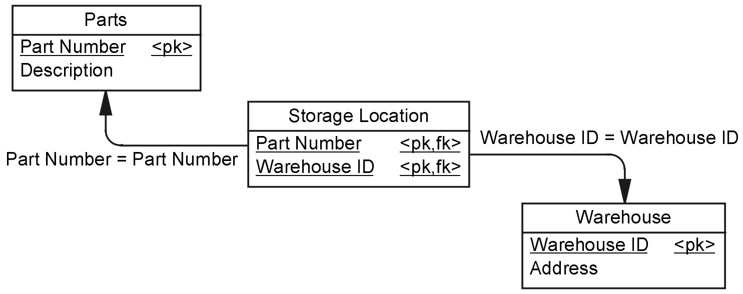
The above entity-relationship model thus resolves the database base structure, below.



- **Many to many** In a many-to-many relationship, a new table is created with two foreign keys. This arrangement is necessary to make the database efficient.

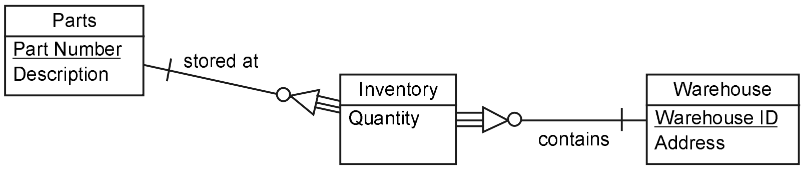


The new Storage Location table relates the Parts and Warehouse tables.

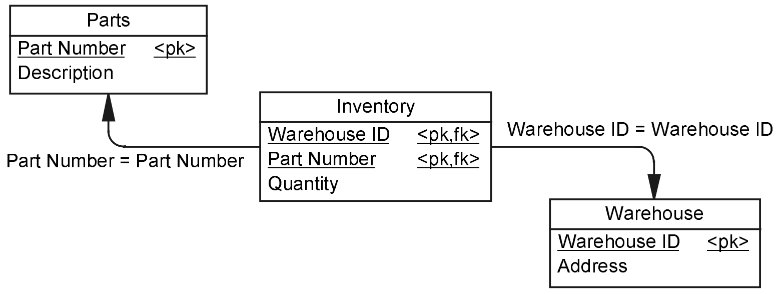


Resolving relationships that carry data

Some of your relationships may carry data. This situation often occurs in many-to-many relationships.



If this is the case, each entity resolves to a table. Each role becomes a foreign key that points to another table.



The Inventory entity borrows its identifiers from the Parts and Warehouse tables, because it depends on both of them. Once resolved, these borrowed identifiers form the primary key of the Inventory table.

Tip
A conceptual data model simplifies the design process because it hides a lot of details. For example, a many-to-many relationship always generates an extra table and two foreign key references. In a conceptual data model, you can usually denote all of this structure with a single connection.

Step 5: Verify the design

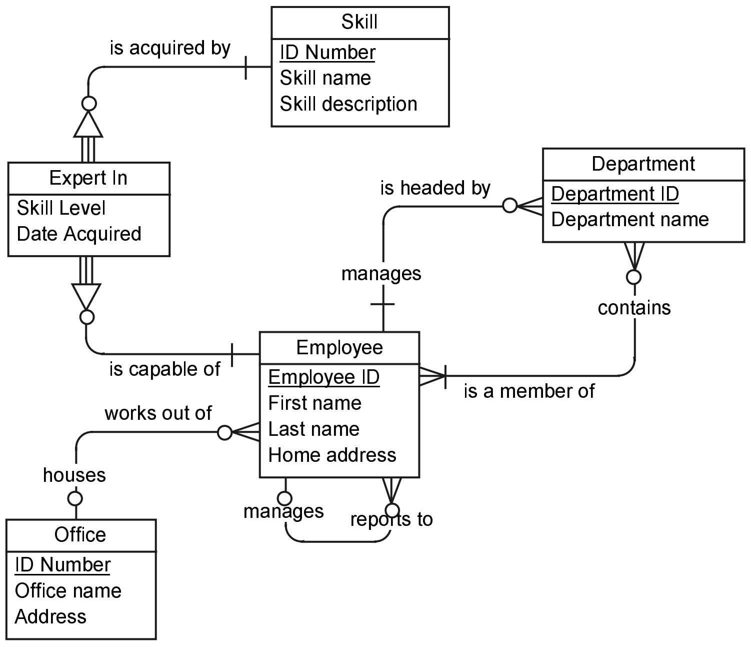
Before you implement your design, you need to make sure that it supports your needs. Examine the activities you identified at the start of the design process and make sure you can access all of the data that the activities require.

- Can you find a path to get the information you need?
- Does the design meet your needs?
- Is all of the required data available?

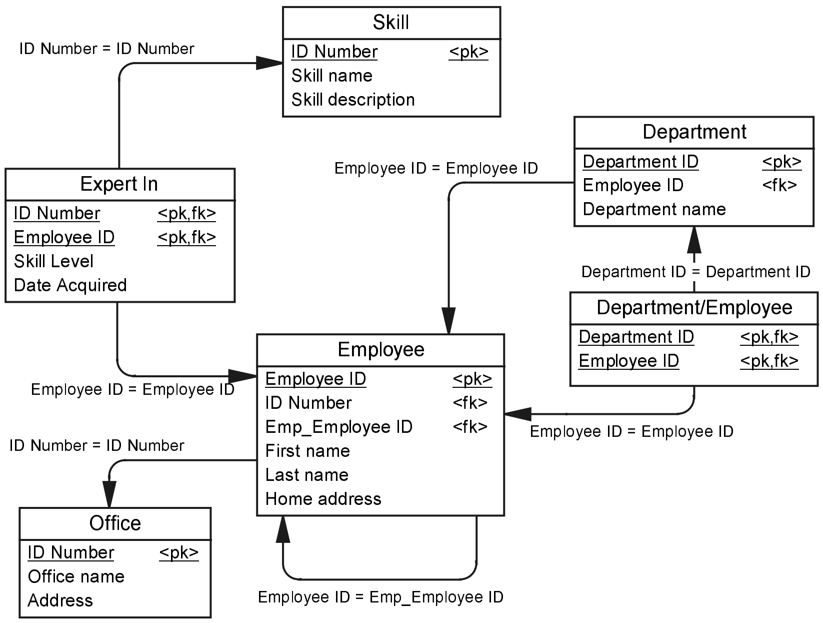
If you can answer yes to all the questions above, you are ready to implement your design.

Final design

Applying steps 1 through 3 to the database for the little company produces the following entity-relationship diagram. This database is now in third normal form.



The corresponding physical data model appears below.



Designing the database table properties

The database design specifies which tables you have and what columns each table contains. This section describes how to specify each column's properties.

For each column, you must decide the column name, the data type and size, whether or not NULL values are allowed, and whether you want the database to restrict the values allowed in the column.

Choosing column names

A column name can be any set of letters, numbers or symbols. However, you must enclose a column name in double quotes if it contains characters other than letters, numbers, or underscores, if it does not begin with a letter, or if it is the same as a keyword.

See "Alphabetical list of keywords" on page 215 of the book *Adaptive Server Anywhere Reference Manual*.

Choosing data types for columns

Available data types in Adaptive Server Anywhere include the following:

- Integer data types
- Decimal data types
- Floating-point data types
- Character data types
- Binary data types
- Date/time data types
- User-defined data types
- Java class data types

For a description of data types, see "SQL Data Types" on page 219 of the book *Adaptive Server Anywhere Reference Manual*.

The data type of the column affects the maximum size of the column. For example, if you specify SMALLINT, a column can contain a maximum value of 32,767. If you specify INTEGER, the maximum value is 2,147,483,647. In the case of CHAR, you must specify the maximum length of a value in the column.

The long binary data type can be used to store information such as images (for instance, stored as bitmaps) or word-processing documents in a database. These types of information are commonly called binary large objects, or BLOBS.

☞ For a complete description of each data type, see "SQL Data Types" on page 219 of the book *Adaptive Server Anywhere Reference Manual*.

NULL and NOT NULL

If the column value is mandatory for a record, you define the column as being NOT NULL. Otherwise, the column is allowed to contain the NULL value, which represents no value. The default in SQL is to allow NULL values, but you should explicitly declare columns NOT NULL unless there is a good reason to allow NULL values.

☞ For a complete description of the NULL value, see "NULL value" on page 213 of the book *Adaptive Server Anywhere Reference Manual*. For information on its use in comparisons, see "Search conditions" on page 194 of the book *Adaptive Server Anywhere Reference Manual*.

Choosing constraints

Although the data type of a column restricts the values that are allowed in that column (for example, only numbers or only dates), you may want to further restrict the allowed values.

You can restrict the values of any column by specifying a CHECK constraint. You can use any valid condition that could appear in a WHERE clause to restrict the allowed values. Most CHECK constraints use either the BETWEEN or IN condition.

☞ For more information about valid conditions, see "Search conditions" on page 194 of the book *Adaptive Server Anywhere Reference Manual*. For more information about assigning constraints to tables and columns, see "Ensuring Data Integrity" on page 347.

Example

The sample database has a table called **Department**, which has columns named **dept_id**, **dept_name**, and **dept_head_id**. Its definition is as follows:

Column	Data Type	Size	Null/Not Null	Constraint
dept_id	integer	—	not null	None
dept_name	char	40	not null	None
dept_head_id	integer	—	null	None

If you specify NOT NULL, a column value must be supplied for every row in the table.