Working with Multiple-Table Queries

3

Most database applications (and *all* well-designed database applications) store their information in multiple tables. Although most of these tables have nothing to do with each other (for example, tables of customer information and employee payroll data), it's likely that at least some of the tables do contain related information (such as tables of customer information and customer orders).

Working with multiple, related tables in a query presents you with two challenges: You need to design your database so that the related data is accessible, and you need to set up links between the tables so that the related information can be retrieved and worked with quickly and easily in the query design window. This chapter tackles both challenges and shows you how to exploit the full multiple-table powers of Access.

IN THIS CHAPTER

Relational Database Fundamentals

Why do you need to worry about multiple tables, anyway? Isn't it easier to work with one large table instead of two or three medium-sized ones? To answer these questions and demonstrate the problems that arise when you ignore relational database models, take a look at a simple example: a table of sales leads.

The Pitfalls of a Nonrelational Design

Table 3.1 outlines a structure of a simple table (named Leads) that stores data on sales leads.

Table 3.1 A Structure of a Simple Sales Leads Table (Leads)

Field	Description
LeadID	The primary key.
FirstName	The contact's first name.
LastName	The contact's last name.
Company	The company that the contact works for.
Address	The company's address.
City	The company's city.
State	The company's state.
Zip	The company's ZIP code.
Phone	The contact's phone number.
Fax	The contact's fax number.
Source	Where the lead came from.
Notes	Notes or comments related to the sales lead.

This structure works fine until you need to add two or more leads from the same company (a not-uncommon occurrence). In this case, you end up with repeating information in the Company, Address, City, and State fields. (The Zip field also repeats, as do, in some cases, the Phone, Fax, and Source fields.)

All this repetition makes the table unnecessarily large, which is bad enough, but it also creates two major problems:

- During data entry, the repeated information must be entered for each lead from the same company.
- If any of the repeated information changes (such as the company's name or address), each corresponding record must be changed.

One way to eliminate the repetition and solve the data entry and maintenance inefficiencies is to change the table's focus. As it stands, each record in the table identifies a specific contact in a company. But it's the company information that repeats, so it makes some sense to allow only one record per company. You can then include separate fields for each sales lead within the company. The new structure might look something like the one shown in Table 3.2.

Table 3.2 A Revised, Company-Centered Structure of the Sales Leads Table

Field	Description
LeadID	The primary key.
Company	The company's name.
Address	The company's address.
City	The company's city.
State	The company's state.
Zip	The company's ZIP code.
Phone	The company's phone number.
Fax	The company's fax number.
First_1	The first name of contact #1.
Last_1	The last name of contact #1.
Source_1	Where the lead for contact #1 came from.
Notes_1	Notes or comments related to contact #1.
First_2	The first name of contact #2.
Last_2	The last name of contact #2.
Source_2	Where the lead for contact #2 came from.
Notes_2	Notes or comments related to contact #2.
First_3	The first name of contact #3.
Last_3	The last name of contact #3.
Source_3	Where the lead for contact #3 came from.
Notes_3	Notes or comments related to contact #3.

In this setup, the company information appears only once, and the contact-specific data (I'm assuming this involves only the first name, last name, source, and notes) appears in separate field groups (for example, First_1, Last_1, Source_1, and Notes_1). This solves the earlier problems, but at the cost of a new dilemma: The structure as it stands will hold only three sales leads per company. Of course, it's entirely conceivable that a large firm might have more than three contacts—perhaps even dozens. This raises two unpleasant difficulties:

- If you run out of repeating groups of contact fields, new ones must be added. Although this might not be a problem for the database designer, most data-entry clerks generally don't have access to the table design (nor should they).
- Empty fields take up as much disk real estate as full ones, so making room for, say, a dozen contacts from one company means that all the records that have only one or two contacts have huge amounts of wasted space.

How a Relational Design Can Help

To solve the twin problems of repetition between records and repeated field groups within records, you need to turn to the relational database model. This model was developed by Dr. Edgar Codd of IBM in the early 1970s. It was based on a complex relational algebra theory, so the pure form of the rules and requirements for a true relational database setup is quite complicated and decidedly impractical for business applications. The next few sections look at a simplified version of the model.

Step 1: Separate the Data

After you know which fields you need to include in your database application, the first step in setting up a relational database is to divide these fields into separate tables where the "theme" of each table is unique. In technical terms, each table must be composed of only entities (that is, records) from a single *entity class*.

For example, the table of sales leads you saw earlier dealt with data that had two entity classes: the contacts and the companies they worked for. Every one of the problems encountered with that table can be traced to the fact that we were trying to combine two entity classes into a single table. So the first step toward a relational solution is to create separate tables for each class of data. Table 3.3 shows the table structure of the contact data (the Contacts table) and Table 3.4 shows the structure of the company information (the Companies table). Note, in particular, that both tables include a primary key field.

Table 3.3 The Structure of the Contacts Table

Field	Description
ContactID	The primary key.
FirstName	The contact's first name.
LastName	The contact's last name.
Phone	The contact's phone number.
Fax	The contact's fax number.
Source	Where the lead came from.
Notes	Notes or comments related to the sales lead.

Table 3.4 The Structure of the Companies Table

Field	Description
CompanyID	The primary key.
CompanyName	The company's name.
Address	The company's address.
City	The company's city.

Table 3.4 Continued

Field	Description
State	The company's state.
Zip	The company's ZIP code.
Phone	The company's phone number (main switchboard).

Step 2: Add Foreign Keys to the Tables

At first glance, separating the tables seems self-defeating because, if you've done the job properly, the two tables will have nothing in common. So the second step in this relational design is to define the commonality between the tables.

In the sales leads example, what is the common ground between the Contacts and Companies tables? It's that every one of the leads in the Contacts table works for a specific firm in the Companies table. So what's needed is some way of relating the appropriate information in Companies to each record in Contacts (without, of course, the inefficiency of simply cramming all the data into a single table, as we tried earlier).

The way you do this in relational database design is to establish a field that is common to both tables. You can then use this common field to set up a link between the two tables. The field you use must satisfy three conditions:

- It must not have the same name as an existing field in the other table.
- It must uniquely identify each record in the other table.
- To save space and reduce data entry errors, it must be the smallest field that satisfies the two preceding conditions.

In the sales leads example, a field needs to be added to the Contacts table that establishes a link to the appropriate record in the Companies table. The CompanyName field uniquely identifies each firm, but it's too large to be of use. The Phone field is also a unique identifier and is smaller, but the Contacts table already has a Phone field. The best solution is to use CompanyID, the Companies table's primary key field. Table 3.5 shows the revised structure of the Contacts table that includes the CompanyID field.

Table 3.5 The Final Structure of the Contacts Table

Field	Description
ContactID	The primary key.
CompanyID	The Companies table foreign key.
FirstName	The contact's first name.
LastName	The contact's last name.
Phone	The contact's phone number.

Table 3.5 Continued

Field	Description
Fax	The contact's fax number.
Source	Where the lead came from.
Notes	Notes or comments related to the sales lead.

When a table includes a primary key field from a related database, the field is called a *foreign key*. Foreign keys are the secret to successful relational database design.

Step 3: Establish a Link Between the Related Tables

After you have your foreign keys inserted into your tables, the final step in designing your relational model is to establish a link between the two tables. This step is covered in detail later in this chapter (see "Establishing Table Relationships").

Types of Relational Models

Depending on the data you're working with, you can set up one of several relational database models. In each of these models, however, you need to differentiate between a *child* table (also called a *dependent* table or a *controlled* table) and a *parent* table (also called a *primary* table or a *controlling* table). The child table is the one that is dependent on the parent table to fill in the definition of its records. The Contacts table, for example, is a child table because it is dependent on the Companies table for the company information associated with each person.

The One-To-Many Model

The most common relational model is one where a single record in the parent table relates to multiple records in the child table. This is called a *one-to-many* relationship. The sales leads example is a one-to-many relationship because one record in the Companies table can relate to many records in the Contacts table (in other words, you can have multiple sales contacts from the same firm). In these models, the "many" table is the one where you add the foreign key.

Another example of a one-to-many relationship is an application that tracks accounts-receivable invoices. You need one table for the invoice data (Invoices) and another for the customer data (Customers). In this case, one customer can place many orders, so Customers is the parent table, Invoices is the child table, and the common field is the Customer table's primary key.

The One-to-One Model

If your data requires that one record in the parent table be related to only one record in the child table, you have a *one-to-one* model. The most common use of one-to-one relations is to

create separate entity classes to enhance security. In a hospital, for example, each patient's data is a single entity class, but it makes sense to create separate tables for the patient's basic information (such as the name, address, and so on) and his or her medical history. This enables you to add extra levels of security to the confidential medical data (such as a password). The two tables then become related based on a common "PatientID" key field.

Another example of a one-to-one model is employee data. You separate the less-sensitive information such as job title and startup date into one table, and restricted information such as salary and commissions into a second table. If each employee has a unique identification number, you use that number to set up a relationship between the two tables.

Note that in a one-to-one model, the concepts of *child* and *parent* tables are interchangeable. Each table relies on the other to form the complete picture of each patient or employee.

The Many-to-Many Model

In some cases, you might have data in which many records in one table can relate to many records in another table. This is called a *many-to-many* relationship. In this case, there is no direct way to establish a common field between the two tables. To see why, let's look at an example from a pared-down accounts-receivable application.

Table 3.6 shows a simplified structure of an Invoices table. It includes a primary key—InvoiceID—as well as a foreign key—CustomerID—from a separate table of customer information (which I ignore in this example).

Table 3.6 The Structure of an Invoices Table

Field	Description
InvoiceID	The primary key.
CustomerID	The foreign key from a table of customer data.

Table 3.7 shows a stripped-down structure of a table of product information. It includes a primary key field—ProductID—and a description field—Product.

Table 3.7 The Structure of a Products Table

Field	Description
ProductID	The primary key.
Product	The product description.

The idea here is that a given product can appear in many invoices, and any given invoice can contain many products. This is a many-to-many relationship, and it implies that *both* tables are parents (or, to put it another way, neither table is directly dependent on the other). But relational theory says that a child table is needed to establish a common field. In this case, the solution is to set up a third table—called a *relation table*—that is the child of both the

original tables. In the ongoing example, the relation table contains the detail data for each invoice. Table 3.8 shows the structure of such a table. As you can see, the table includes foreign keys from both Invoices (InvoiceID) and Products (ProductID), as well as a Quantity field.

Table 3.8 The Structure of a Table of Invoice Detail Data

Field	Description
InvoiceID	The foreign key from the Invoices table.
ProductID	The foreign key from the Products table.
Quantity	The quantity ordered.

Enforcing Referential Integrity

Database applications that work with multiple, related tables need to worry about enforcing *referential integrity rules*. These rules ensure that related tables remain in a consistent state relative to each other. In the sales leads application, for example, suppose the Companies table includes an entry for "ACME Coyote Supplies" and that the Contacts table contains three leads who work for ACME. What would happen if you deleted the ACME Coyote Supplies record from the Companies table? Well, the three records in the Contacts table would no longer be related to any record in the Companies table. Child records without corresponding records in the parent table are called, appropriately enough, *orphans*. This situation leaves your tables in an inconsistent state, which can have unpredictable consequences.

Preventing orphaned records is what is meant by enforcing referential integrity. You need to watch out for two situations:

- Deleting a parent table record that has related records in a child table.
- Adding a child table record that isn't related to a record in the parent table (either because the common field contains no value or because it contains a value that doesn't correspond to any record in the parent table).

Establishing Table Relationships

Now that you know the theory behind the relational model, you can turn your attention to creating and working with related tables in queries. The first step, however, is to establish the relationship between the two tables, which is what this section is all about.

To get started, choose Tools, Relationships (or click the Relationships button on the Database toolbar). You'll see the Relationships window, shown in Figure 3.1. (Note that you'll see this view of the window only if you're working with the Northwind sample database.)

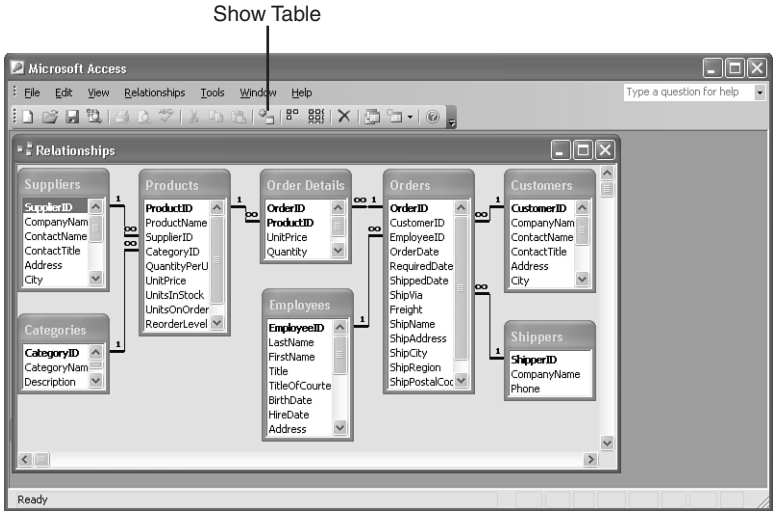


Figure 3.1
You use the Relationships window to establish relations between tables.

Understanding Join Lines

Because the Northwind sample database is well-designed, all the tables are related to each other in one way or another. You can tell this by observing the lines that connect each table in the Relationships window. These lines are called *join lines*. As you can see in Figure 3.1, the join line connects the two fields that contain the related information. For example, the Suppliers and Products tables are joined on the common SupplierID field. In this case, SupplierID is the primary key field for the Suppliers table, and it appears as a foreign key in the Products table. This lets you relate any product to its corresponding supplier data.

The symbols attached to the join lines tell you the type of relation. In the join between the Suppliers and Products tables, for example, the Suppliers side of the join line has a 1, and the Products side of the line has an infinity symbol (8). This stands for "many," so you interpret this join as a one-to-many relation.

Types of Joins

Access lets you set up four kinds of joins:

Inner join—An *inner join* includes only those records in which the related fields in the two tables match each other exactly (which is why this type of join is often called an *equijoin*). This is the most common type of join.

Outer join—An *outer join* includes every record from one of the tables and only those records from the other table in which the related fields match each other exactly. In your sales leads example, it's possible that there might be companies for which no contacts have yet been established. Creating an inner join between the Company and Contacts table shows you only those firms that have existing contacts. However, setting up an outer join shows *all* the records in the Companies table, including those in which there is no corresponding record in the Contacts table.

> **NOTE**
>
> An outer join is also called a *left-outer join*. To see why, consider a one-to-many relation. Here, the "left" side is the "one," table and the "right" side is the "many" table. So this type of join includes every record from the "one" (left) side and only those matching records from the "many" (right) side.
>
> You use the term *left-outer join* when you need to differentiate it from a *right-outer join*. In a one-to-many relation, this type of join includes every record from the "many" (right) side and only those matching records from the "one" (left) side.

Self-join—A *self-join* is a join on a second copy of the same table. Self-joins are handy for tables that include different fields with the same type of information. For example, the Northwind Employees table has an EmployeeID field that lists the identification number of each employee. The same table also includes a ReportsTo field that lists the identification number of the employee's manager. To display the name of each employee's manager, you use a second copy of the Employees table and join the EmployeeID and ReportsTo fields.

Theta join—A *theta join* is created when the data in two fields from two tables is related via some comparison operator other than equals (=). For example, a *not-equal join* relates data using the not equal operator (<>). For example, suppose you want to compare the unit price data in Northwind's Order Details table with the unit price data in the Products table. Specifically, you want to see those orders where the unit price of the order differs from the unit price of the product. In this case, you look for records where the [Order Details].UnitPrice field is not equal to the [Products].UnitPrice field.

Adding Tables to the Relationships Window

If you need to establish a new relationship between two tables, your first order of business is to add the tables to the Relationships window. Here are the steps to follow:

1. Choose Relationships, Show Table (or click the Show Table button on the toolbar). Access displays the Show Table dialog box, shown in Figure 3.2.

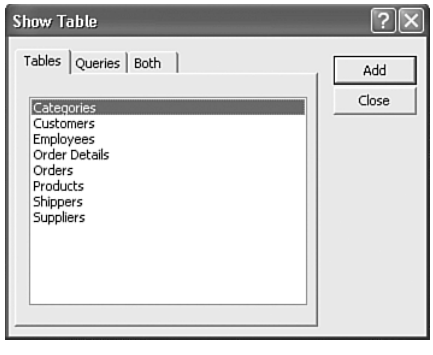


Figure 3.2
Use this dialog box to add tables to the Relationships window.

2. Click the table you want to add.
3. Click Add. Access adds the table to the Relationships window.
4. Repeat steps 2 and 3 to add more tables.
5. Click Close to return to the Relationships window.

Joining Tables

To create a join between two tables (or a self-join between two copies of the same table), use the mouse to click and drag one of the related fields and drop it on the other. Here are the specific steps:

1. Add the tables you want to join.
2. Arrange the table boxes so that in each box you can see the fields you want to use for the join.
3. Click and drag the related field from one table and drop it on the related field in the other table. Access displays the Edit Relationships dialog box, shown in Figure 3.3.

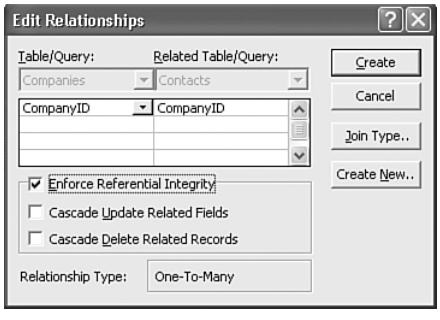


Figure 3.3
Access displays the Edit Relationships dialog box when you click and drag a related field from one table and drop it on another.

4. The grid should show the names of the fields in each table that you want to relate. If not, use the drop-down list in one or both cells to click the correct field or fields.
5. If you want Access to enforce referential integrity rules on this relation, click the Enforce Referential Integrity check box. If you do this, two other check boxes become active:

   Cascade Update Related Fields—If you click this check box and then make changes to a primary key value in the parent table, Access updates the new key value for all related records in all child tables. For example, if you change a CompanyID value in the Companies table, all related records in the Contacts table have their CompanyID fields updated automatically.

   Cascade Delete Related Fields—If you activate this check box and then delete a record from the parent table, all related records in all child tables are also deleted. For example, if you delete a record from the Companies table, all records in the Contacts table that have the same CompanyID as the deleted record are also deleted.

6. To set the type of join, choose Join Type to display the Join Properties dialog box, shown in Figure 3.4. Here, option 1 corresponds to an inner join, option 2 corresponds to a left-outer join, and option 3 corresponds to a right-outer join. When you've clicked the option you want, click OK to return to the Relationships dialog box.

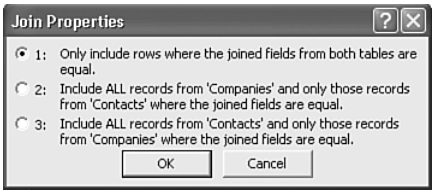


Figure 3.4
Use the Join Properties dialog box to establish the type of join.

7. Click Create. Access establishes the relationship and displays a join line between the two fields.

Editing a Relationship

If you need to make changes to a relationship, Access lets you edit the relation parameters from within the Relationships window. For the relation you want to adjust, click the join line for the two fields and then choose Relationships, Edit Relationship (you can also right-click the join line and choose Edit Relationship from the shortcut menu). Access displays the Edit Relationships dialog box so that you can make your changes.

Removing a Join

If you no longer need a join, you can remove it by clicking the join line and choosing Edit, Delete (or by pressing Delete). When Access asks you to confirm the deletion, choose Yes.

Working with Multiple Tables in a Query

With a properly constructed relational database model, you'll end up with fields that don't make much sense by themselves. For example, the Northwind database has an Order Details table that includes a ProductID field—a foreign key from the Products table. This field contains only numbers and therefore by itself is meaningless to an observer.

The idea behind a multiple-table query is to *join* related tables and by doing so create a dynaset that replaces meaningless data (such as a product ID) with meaningful data (such as a product name).

The good news is that after you've established a relationship between two tables, Access handles everything else behind the scenes, so working with multiple tables isn't much harder than working with single tables.

Adding Multiple Tables to a Query

To add multiple tables to a query, follow these steps:

1. Display the Show Table dialog box. You have two choices:
   - If you're starting a new query without highlighting a table in advance, wait until you see the Show Table dialog box onscreen.
   - If you're already in the query design window, choose Query, Show Table (or click the toolbar's Show Table button).
2. Click the table name and then choose Add.
3. Repeat step 2 to add other tables, as necessary.
4. Click Close.

As you can see in Figure 3.5, Access displays join lines between related tables.

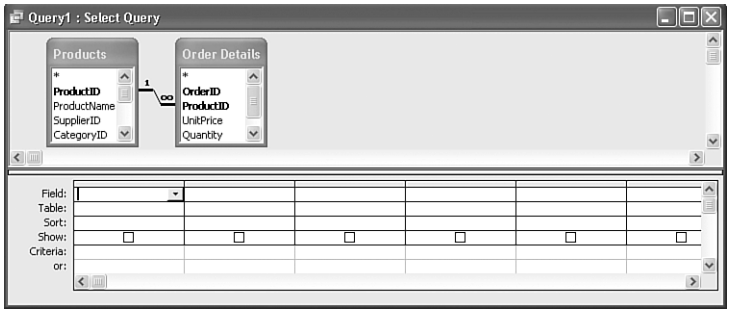


Figure 3.5
When you add multiple, related tables to the query design window, Access automatically displays the join lines for the related fields.

Adding Fields from Multiple Tables

With your tables added to the query design window, adding fields to the query is only slightly different than adding them for a single-table query:

- You can still add any field by clicking and dragging it from the table pane to one of the `Field` cells in the design grid.
- When you choose a field directly from a `Field` drop-down list, note that the field names are preceded by the table name (for example, `Products.SupplierID`).
- To lessen the clutter in the `Field` cells, first use the `Table` cell to choose the table that contains the field you want. After you do this, the list in the corresponding `Field` cell will display only the fields from the selected table.

From here, you can set up the query criteria, sorting, top *N* values, and calculated columns exactly as you can with a single-table query. Figure 3.6 shows a query based on the Products, Order Details, and Orders tables. The query shows the SupplierID, ProductName, and UnitsInStock (from Products), the Quantity (from Order Details), and OrderDate (from Orders), and a Left In Stock calculated column that subtracts the

Quantity from the UnitsInStock. The dynaset will contain just those orders from May 6, 1998 and is sorted on the LeftInStock calculated column. Figure 3.7 shows the resulting dynaset.

Figure 3.6
A query with three related tables that includes fields from all the tables.

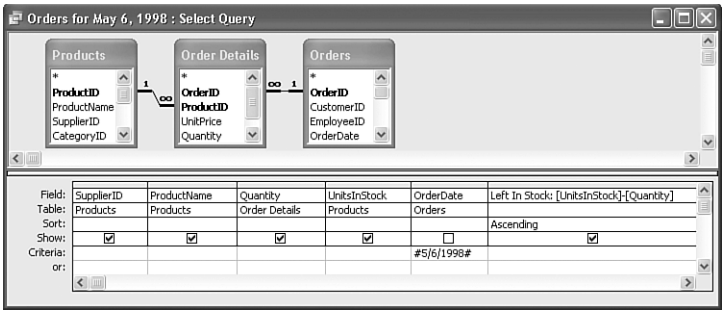
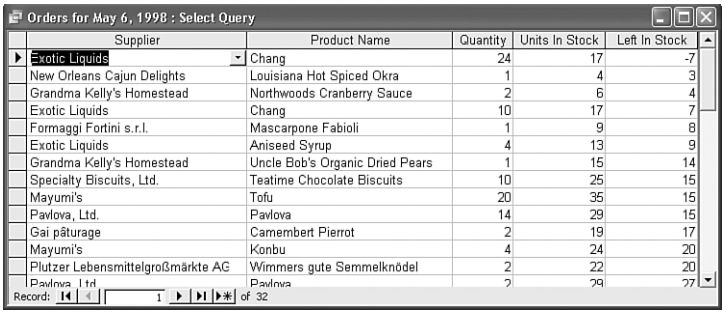


Figure 3.7
The dynaset returned by the multiple-table query shown in Figure 3.6.



Supplier	Product Name	Quantity	Units In Stock	Left In Stock
Exotic Liquids	Chang	24	17	-7
New Orleans Cajun Delights	Louisiana Hot Spiced Okra	1	4	3
Grandma Kelly's Homestead	Northwoods Cranberry Sauce	2	6	4
Exotic Liquids	Chang	10	17	7
Formaggi Fortini s.r.l.	Mascarpone Fabioli	1	9	8
Exotic Liquids	Aniseed Syrup	4	13	9
Grandma Kelly's Homestead	Uncle Bob's Organic Dried Pears	1	15	14
Specialty Biscuits, Ltd.	Teatime Chocolate Biscuits	10	25	15
Mayumi's	Tofu	20	35	15
Pavlova, Ltd.	Pavlova	14	29	15
Gai pâturage	Camembert Pierrot	2	19	17
Mayumi's	Konbu	4	24	20
Plutzer Lebensmittelgroßmärkte AG	Wimmers gute Semmelknödel	2	22	20
Pavlova, Ltd.	Pavlova	2	29	27

Record: 1 of 32

NOTE

In the query shown in Figure 3.6, the Products and Orders table are said to have an *indirect relationship*. That is, they're related to each other, but only via the Order Details table. Note that it's possible to construct a query that only includes fields from Products and Orders, but you must still include the Order Details table in the query design to allow Access to set up the indirect relationship.

The only thing you have to watch out for is dealing with tables that each have a field with the same name. For example, both the Order Details table and the Products table have a UnitPrice field. To differentiate between them in, say, an expression for a calculated column, you need to preface the field name with the table name, like so:

```
[Table Name].[FieldName]
```

For example, consider the formula that calculates the ExtendedPrice field in the Order Details Extended query. The idea behind this formula is to multiply the unit price times the quantity ordered and subtract the discount. Here's the formula:

```
[Order Details].[UnitPrice]*[Quantity]*(1-[Discount])
```

To differentiate between the UnitPrice field in the Order Details table and the UnitPrice field in the Products table, the formula uses the term [Order Details].[UnitPrice], as shown in Figure 3.8.

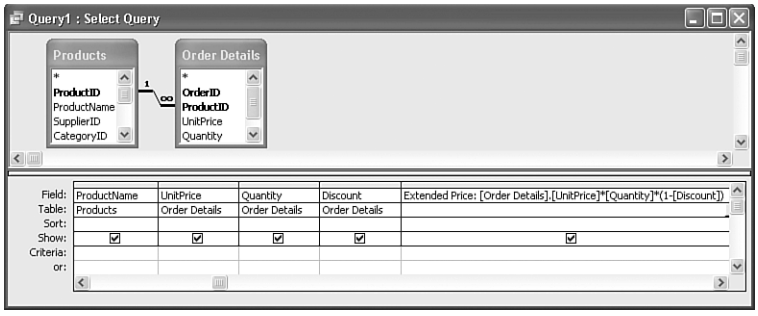


Figure 3.8
When the tables in a multiple-table query share a common field name, precede the field name with the table name in an expression.

Nesting Queries Within Queries

In the previous section I showed you how to add multiple tables to a query, but there's no reason why you can't also add other queries to the table pane. After all, the Show Table dialog box has a Queries tab that lists all your saved queries, so you can add them to the query design as easily as you add a table. When you nest one query inside another, Access runs the nested query first, and then uses the resulting dynaset to produce the rest of the query.

For example, in the Orders for May 6, 1998 query from the previous section, the Left In Stock calculated column returned the number of units each product had left in stock after subtracting the order quantity. Suppose you then wanted a new query that checked for those products with a negative Left In Stock value and returned the appropriate supplier data so that the product could be reordered.

To do this, you begin by adding both the Suppliers table and the Orders for May 6, 1998 query to the query design window. As you can see in Figure 3.9, Access automatically sets up a temporary relation between the common SupplierID fields (because SuppliersID is the primary key of the Suppliers table). The rest of the query is created in the usual way, and the result is shown in Figure 3.10.

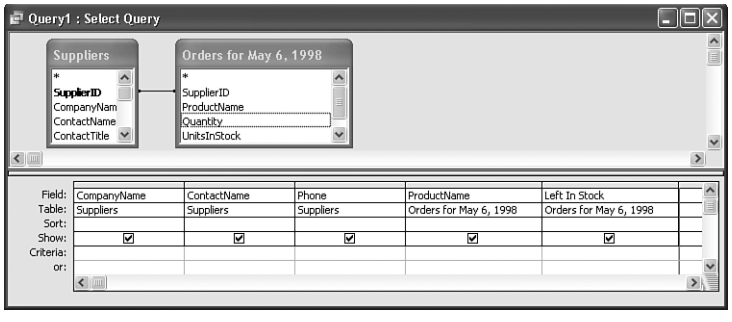
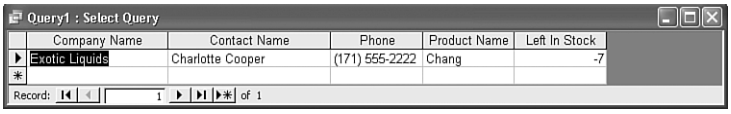


Figure 3.9
You can nest one query inside another and Access will set up a temporary relation based on a common field.

Figure 3.10
The dynaset produced by the query in Figure 3.9.



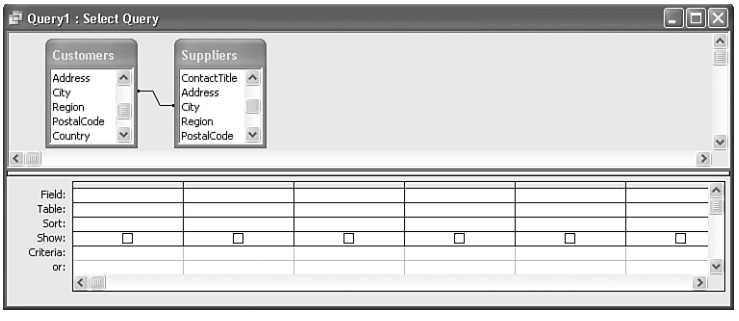
Joining Tables Within the Query Design Window

As you've seen, Access recognizes existing relationships in multiple-table queries, and will also set up a temporary relationship if the common field is a primary key in one of the tables.

In other cases, you can have fields with common or similar data, but there's no existing relationship and Access doesn't set up a temporary relationship. For example, both Northwind's Customers and Suppliers tables have a City field. How can you create an inner join on this common field (to, for example, see which customers and suppliers are located in the same city)?

You can do this by creating a temporary relationship. After you've added the tables to the query design window's table pane, click and drag the field from one table and drop it on the related field in the other table. Access displays a temporary join line between the fields, as shown in Figure 3.11.

Figure 3.11
To create a temporary relationship by hand, click and drag the field from one table and drop it on the related field in the other table.



Creating Other Types of Joins

So far you've worked only with inner joins, which is as it should be because inner joins are by far the most common, particularly in a business environment. However, the three other types of joins—outer, self, and theta—can also come in handy and are discussed in the next three sections.

Creating Outer Joins

An outer join is one where *all* the records in one table are included in the dynaset regardless of whether there are matching records in the other table. For example, suppose you're dealing with Northwind's Customers and Orders tables, which are related on the common CustomerID field. An inner join between these tables shows only those customers who have

placed orders. By contrast, an outer join on the Customers table displays all the records from that table, even customers who have never placed an order.

There are two types of outer joins:

Left outer join—This join displays all the records from the "left" table. For example, in tables with a one-to-many relationship, the left outer join displays all the records from the "one" table.

Right outer join—This join displays all the records from the "right" table. For example, in tables with a one-to-many relationship, the left outer join displays all the records from the "many" table.

To set the type of outer join, follow these steps:

1. Add the tables to the query design window.
2. Create the relationship between the tables, if one doesn't exist.
3. Choose View, Join Properties. Access displays the Join Properties dialog box, shown in Figure 3.12.

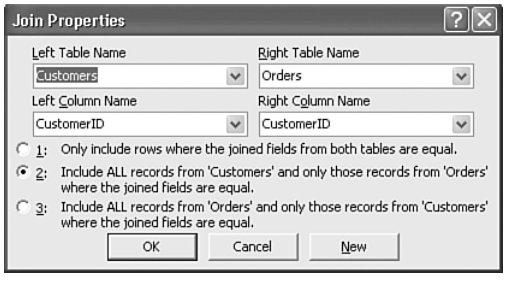


Figure 3.12
Use the Join Properties dialog box to select the type of join you want.

4. Option 1 creates an inner join. To change to an outer join, click either 2 (for a left outer join) or 3 (for a right outer join).
5. Click OK.

Using Outer Joins to Find Records Without Matching Records in a Related Table

The most common use for outer joins is to look for records in one table that don't have a matching record in some related table. For example, you can look for records in the Customers table that have no corresponding records in the Orders table; this tells you which customers have not yet placed orders. Similarly, you can look for records in the Products table that have no corresponding records in the Categories table; this tells you that you have a data entry problem because all products should have a category.

As a general rule, to see only those records without matching records in a related table, do one of the following:

- To see records in the parent table without matching records in the child table, create a left outer join and filter the dynaset by adding `Is Null` as the criteria for the common field in the child table.
- To see orphan records in the child table (that is, records in the child table without any corresponding records in the parent table), create a right outer join and filter the dynaset by adding `Is Null` as the criteria for the common field in the parent table.

The next two sections take you through examples of these techniques.

Finding Customers Without Matching Orders

For example, suppose you want to see a list of customers who haven't yet placed an order. This means you want to join the Customers table and the Order table, which are related on the CustomerID field. You start by displaying all the Customers. This means, because Customers is the parent of Orders, you need to create a left outer join. Figure 3.13 shows the query setup, including the fact that option 2 (left outer join) is chosen in the Join Properties dialog box. Figure 3.14 shows the resulting dynaset.

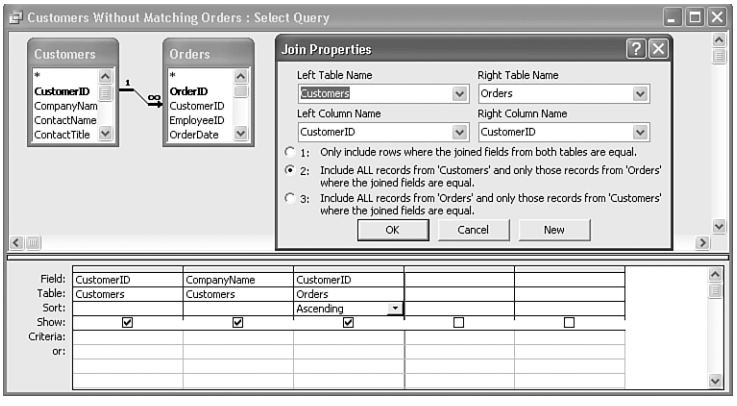


Figure 3.13
A query set up for a left outer join between the Customers and Orders tables.

In Figure 3.14, notice that the first two records in the Custom field is blank. This tells you that these are the records in Customers that have no matching records in Orders, meaning they haven't yet placed any orders. Therefore, rather than displaying all the records, filter the dynaset to show only those where the CustomerID field of the Orders table is equal to `Null`. Figure 3.15 shows the revised query that adds this criterion.

Figure 3.14
The dynaset created by the query in Figure 3.13.

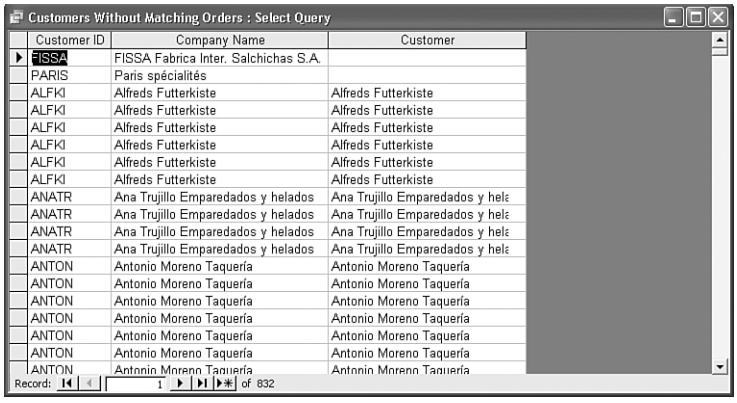
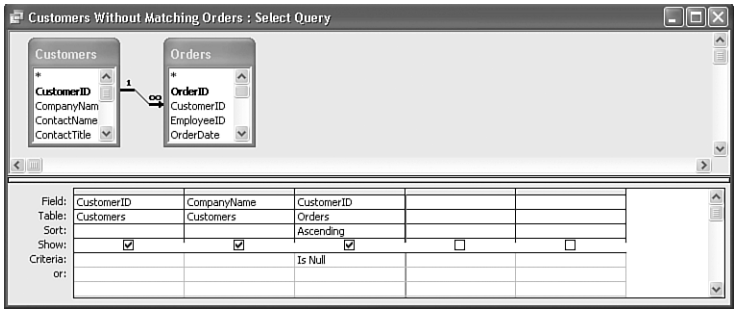


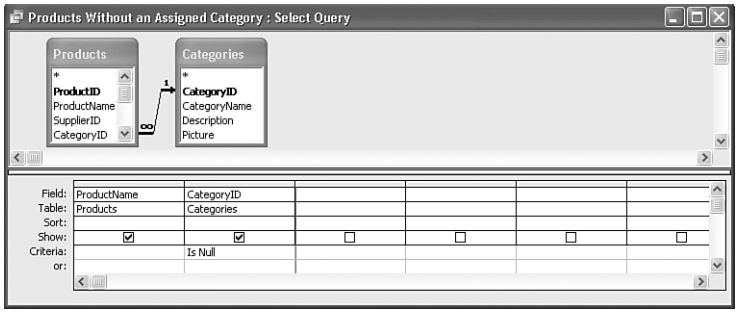
Figure 3.15
To see only those customer without matching orders, add the expression `Is Null` to the Orders.CustomerID field.



Finding Products Without an Assigned Category

In the Products table, each record should have been assigned an item from the Categories table. To make sure, you can build a query that looks for those Products without a matching category. This requires joining the Products table and the Categories table, which are related via the CategoryID field. Because Products is a child of Categories, you need to create a right outer join. You then add the CategoryID field from the Categories table and filter it using the `Is Null` criterion, as shown in Figure 3.16.

Figure 3.16
To find those products without an assigned category, create a right outer join and filter the Categories.CategoryID field using the `Is Null` expression.



Creating Self-Joins

Database tables are sometimes *self-referential*, which means they contain a field with data that points to another field in the same table. A good example is the Northwind Employees table, which includes an EmployeeID field, the primary key that contains the employee identification numbers. Employees also contains the ReportsTo field, which contains the identification number of the person each employee reports to. In other words, each value in the ReportsTo field will have a corresponding value in the EmployeeID field.

If you want to know, for example, which employees have people reporting to them, you need to create a self-join—a table joined to itself—on the Employees table. Creating a self-join involves the following steps:

1. Start a new query and add the table (Employees in this case) *twice*.
2. Create a temporary join by clicking and dragging the field that contains the data (EmployeeID) to the field that contains the subset of the data (ReportsTo).
3. Add the fields you want to use for the query to the design grid and then set up your criteria, sorting, and other query elements.
4. For a self-join to work properly, you need to tell Access to return only unique values in the query. To do this, click an empty spot inside the query design window and then choose View, Properties (or press Alt+Enter). In the Query Properties window, click Yes in the Unique Values list and then close the window.

➔ For more about the Unique Values property, **see** "Creating a Unique Values Query," **p. 86.**

Figure 3.17 shows a self-join on the Employees table, and Figure 3.18 shows the resulting dynaset, which displays the employees who have people reporting to them.

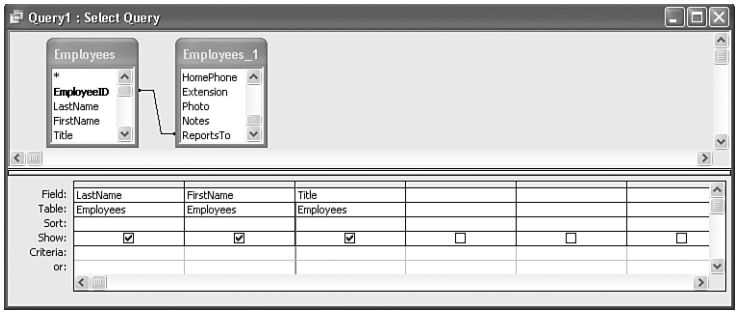


Figure 3.17
A query set up for a self-join on the Employees table.

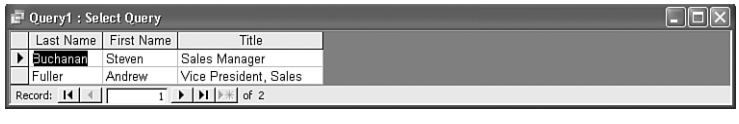


Figure 3.18
The dynaset created by the query in Figure 3.17.

Creating Theta Joins

The joins you've seen so far have all worked on the premise that the join is based on the equality between two fields. In an inner join, for example, you only see records where the joined fields from both tables are equal; similarly, in an outer join, you see all the records from one table, but only those records from the second table where the joined fields are equal.

In business, however, it's sometimes the case that you need a join that's based on fields that are unequal. For example, Northwind's Customers table has a CompanyName field, and its Orders table has a ShipName field. In most cases, these values should be the same; that is, if a customer places an order, that order should be sent to that company. If the shipping name isn't the same as the customer name, it might mean either that the order was sent to the wrong company or that the company name is wrong in one table or the other. (It's also possible that the order is correct and that the customer asked for the shipment to be sent to a different ship address.)

To check into this type of scenario, you need a *not-equal join* that joins two tables and shows only those records where the joined fields from both tables are not equal; for example, joining the Customers and Orders tables based on whether the CompanyName and ShipName fields are not equal.

Here's the procedure to follow to create a not-equal join:

1. Start a new query and add the tables you want to work with (such as Customers and Orders).
2. If no relation exists between the tables, create a temporary join by clicking and dragging the appropriate field from one table and dropping it on the related field in the other table.
3. Add the fields you want to use for the query to the design grid and then set up your criteria, sorting, and other query elements. Be sure to include the fields on which the not-equal join will be based (such as CompanyName from the Customers table and ShipName from the Orders table).
4. In the `Criteria` cell of the field for which you want to check for not-equal values (such as the Orders table's ShipName field), enter a comparison formula using the following general form:

   ```
   <>[RelatedTable].[JoinedField]
   ```

   Here, *`RelatedTable`* is the name of the other table in the query, and *`JoinedField`* is the field from the other table that is joined to the current field. Here's an example for the Orders.ShipName field:

   ```
   <>[Customers].[CompanyName]
   ```

Figure 3.19 shows a not-equal join between the Customers table and the Orders table, with the not-equal criterion added for the Orders.ShipName field. Figure 3.20 shows the

resulting dynaset, which displays the orders where the shipping name is different from the customer name. Notice that query has caught two subtle errors:

- For the customer "Galeria del gastrónomo," the accent is in the wrong place in the ShipName field ("Galeria del gastronómo").
- For the customer "Wolski Zajazd," the CompanyName field has two spaces between "Wolski" and "Zajazd."

Figure 3.19
A query set up for a not-equal join on the Customers and Orders table to look for orders where the ShipName is not equal to the CompanyName.

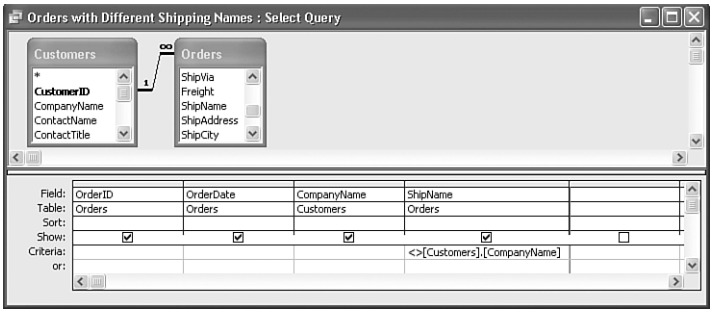


Figure 3.20
The dynaset created by the query in Figure 3.19.

Orders with Different Shipping Names : Select Query

Order ID	Order Date	Company Name	Ship Name
10926	05-Mar-1998	Galería del gastrónomo	Galería del gastronómo
10887	13-Feb-1998	Galería del gastrónomo	Galería del gastronómo
10568	13-Jun-1997	Galería del gastrónomo	Galería del gastronómo
10426	27-Jan-1997	Galería del gastrónomo	Galería del gastronómo
10366	28-Nov-1996	Galería del gastrónomo	Galería del gastronómo
10260	19-Jul-1996	Old World Delicatessen	Ottilies Käseladen
10808	01-Jan-1998	Princesa Isabel Vinhos	Old World Delicatessen
10249	05-Jul-1996	Tradição Hipermercados	Toms Spezialitäten
10248	04-Jul-1996	Wilman Kala	Vins et alcools Chevalier
10998	03-Apr-1998	Wolski Zajazd	Wolski Zajazd
10374	05-Dec-1996	Wolski Zajazd	Wolski Zajazd
10792	23-Dec-1997	Wolski Zajazd	Wolski Zajazd
11044	23-Apr-1998	Wolski Zajazd	Wolski Zajazd
10906	25-Feb-1998	Wolski Zajazd	Wolski Zajazd
10611	25-Jul-1997	Wolski Zajazd	Wolski Zajazd
10870	04-Feb-1998	Wolski Zajazd	Wolski Zajazd
AutoNumber)			

Record: 1 of 16

Creating a Unique Values Query

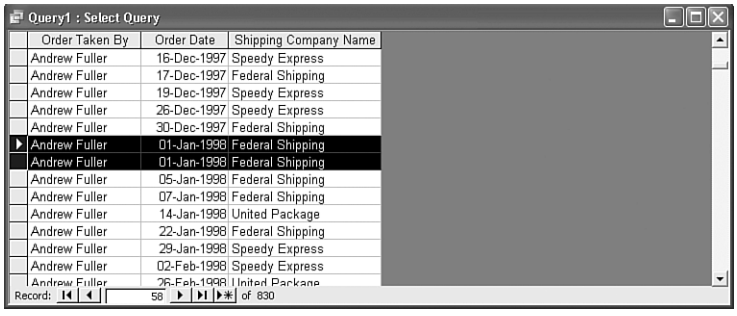
Most business data—dealing as it does with unique customers, suppliers, products, shippers, and employees—is stored in tables that are designed to prevent duplicate records. In most cases, preventing duplicates is a straightforward matter of adding a primary key field (such as a customer account number or employee ID number), which, by definition, does not allow duplicate values. You can also add an index to any field and specify the `Yes (No Duplicates)` option in the table design window's Indexed property.

However, some tables are set up to allow duplicate values. For example, consider a simple table that contains only employee names and the dates on which they took customer orders.

A single employee can take more than one order on a given day, so that table will have multiple records with the same data.

It's also not unusual for a query to return a dynaset that contains duplicate data. For example, suppose you put together a multiple-table query using Northwind's Employees, Orders, and Shippers tables. If you include the Orders table's OrderID in the query, there will be no duplicate records in the dynaset because OrderID is the primary key of the Orders table. However, if you don't include the OrderID field, the dynaset will have duplicate records. Figure 3.21 shows such a dynaset, and you can see that the two highlighted records are the same.

Figure 3.21
This query combines the Employees, Orders, and Shippers tables and, because the OrderID primary key is not part of the dynaset, the result contains duplicate records.



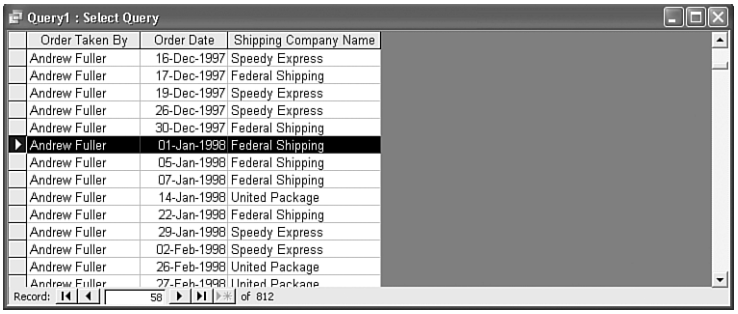
If your multiple-table dynaset is returning what appear to be duplicate records, or if you're working with a single table that contains duplicate records, you can tell Access to include only unique records in the result. You do this by specifying that Access first examine the dynaset data to look for records that have *the same values in all the dynaset fields*, and then display only the first of those records. Because the determination whether a record is a duplicate is based on the field values, Access calls this a *unique values* query.

Follow these steps to create a unique values query:

1. Start a new query and add the tables you want to work with.
2. Add the fields you want to use for the query to the design grid and then set up your criteria, sorting, and other query elements.
3. Click an empty spot inside the query design window and then choose View, Properties (or press Alt+Enter). In the Query Properties window, click Yes in the Unique Values list and then close the window.

When you run the query, Access will display only unique records in the dynaset. In Figure 3.22, for example, you can see that only one of the duplicate records highlighted in Figure 3.21 appears in this unique values version of the query.

Figure 3.22
This is the same query as the one in Figure 3.21, except that the Unique Values property has been set to Yes, thus eliminating the duplicate records in the dynaset.



NOTE

What about the Unique Records property in the Query Properties dialog box? This setting isn't all that useful because it applies only to a very limited case. Suppose you have two tables that are joined via a one-to-many relationship. By definition, data from the "one" table can appear multiple times in the "many" table. So if you then set up a query that uses both tables but includes only fields from the "one" table, there's a good chance the dynaset will display duplicate records.

(Why might you add two tables to a query and then not include any fields from one of the tables in the dynaset? The most common reason is that you're using one or more fields from the second table to enter the criteria for the query, but you don't need to see those fields in the result.)

To suppress the display of duplicate records from the "one" table, change the value of the Unique Records property to Yes.

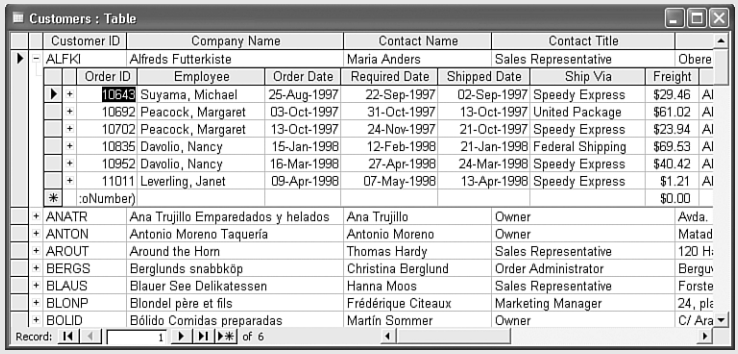
CASE STUDY

Drilling Down to the Order Details

You might know that Access 2003 has the welcome capability to view data contained in another, related table from within the table datasheet. In Figure 3.23, for example, you can see that the Customers table also includes a column of plus signs (+) on the left. Clicking a plus sign displays a new datasheet—called a *subdatasheet*—that, in this case, contains that customer's data from the Orders table (see Figure 3.24).

Figure 3.23
When you view certain tables in the datasheet view, the leftmost column will contain a plus sign (+) for each record.

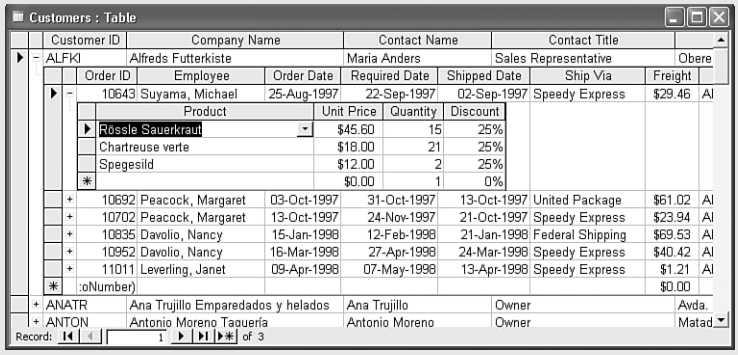
Figure 3.24
Clicking a plus sign displays a subdatasheet showing the related data from another table.



Access sets up this subdatasheet capability whenever you establish a relationship between two tables. The Customers and Orders tables have a one-to-many relationship on the CustomerID field, so clicking a customer's plus sign displays the related records from the Orders table.

Notice, too, that the Orders subdatasheet also has a column of plus signs. Orders has a one-to-many relation with the Order Details table on the OrderID field. So clicking the plus sign beside an order displays the related details, as shown in Figure 3.25.

Figure 3.25
Clicking a plus sign in the Orders subdatasheet displays another subdatasheet that contains the related order details.



The subdatasheet is a great feature, but it suffers from the same problems inherent in all table-based datasheet views: the subdatasheet might show more fields than you need to see, and it might not show fields you want to see (such as the extended price in the order details).

You can solve these problems by using query-based subdatasheets. Because the subdatasheets display data from a query dynaset, you can configure the query to display the data any way you want, and even include calculated columns.

To demonstrate the power and flexibility of query-based subdatasheets, this case study remakes the Customers-Orders-Order Details example to create a more useful way of drilling down into a customer's order details.

Adding a Subdatasheet to a Query

Before getting to the details of the case study, you need to know how to add a subdatasheet to a query. Datasheets and subdatasheets have the same parent-to-child relationship as linked tables. Here are the steps to follow:

1. In the query design window, click an empty spot and then choose View, Properties (or press Alt+Enter). Access displays the Query Properties dialog box.
2. In the Subdatasheet Name list, click the name of the table or query you want to use as the basis for the subdatasheet.
3. In the Link Child Fields box, enter the name of a field from the child table or query (the object you chose in the Subdatasheet Name list). This must be the field that will be related to a field in the parent (which is, in this case, the dynaset created by the query).
4. In the Link Master Fields box, enter the name of a field from the parent query. This must be the field that will be related to the field you entered in step 3.
5. (Optional) Use the Subdatasheet Height box to enter the maximum height, in inches of the subdatasheet. If you enter **0**, Access displays all the records in the subdatasheet.
6. When the Subdatasheet Expanded property is set to No, Access hides all the subdatasheets and you need to click the plus signs (+) to see them; if you prefer that Access display all the subdatasheets automatically, change this property to Yes. Figure 3.26 shows an example of the Query Properties dialog box with these fields filled in.
7. Close the window.

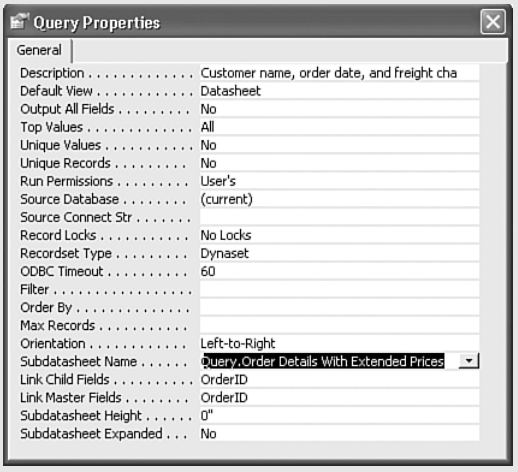


Figure 3.26
To add a subdatasheet to a query, fill in the Subdatasheet Name, Link Child Fields, and Link Master Fields properties.

Working with Query Subdatasheets

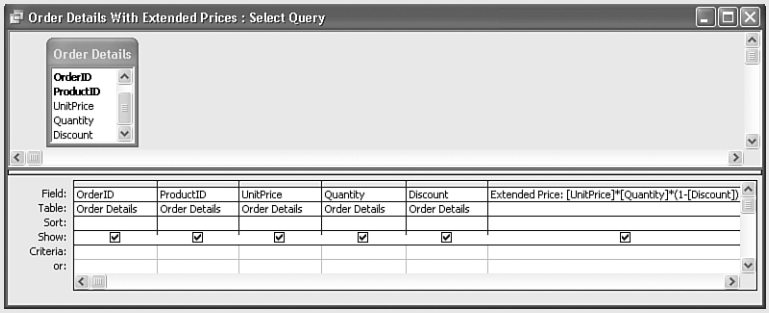
The secret to working with query-based subdatasheets is to work from the bottom up. That is, you begin with the lowest level of data and create a query that displays the data in exactly the way you want. Then you move up to the next level, which will be a query based on a table that has a common field (or, at least, a relatable field) with the first table. Repeat this until you get to the topmost query.

For example, in the Customers-Orders-Order Details case, you proceed as follows:

1. Create a query based on the Order Details table. In this case study, I created a query named Order Details With Extended Prices that shows not only the details such as product, price, quantity, and discount, but also the *extended price* for each product, which is given by a calculated column based on the following expression, as shown in Figure 3.27:

```
[UnitPrice] * [Quantity] * (1 - [Discount])
```

Figure 3.27
The query from the lowest level of the drill-down: the Order Details with an Extended Price calculated column.



2. Create a query based on the Orders table and include the Order Details With Extended Prices query as a subdatasheet (as described in the previous section) related on the OrderID field. For the case study, I named this new query Orders With Extended Order Details.
3. Create a query based on the Customers table and include the Orders With Extended Order Details query as a subdatasheet related on the CustomerID field, as shown in Figure 3.28.

Figure 3.28
The query based on the Customers table also has a subdatasheet that used the Orders With Extended Order Details query.

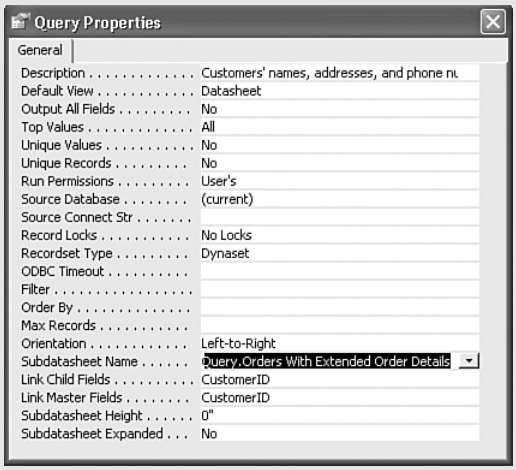
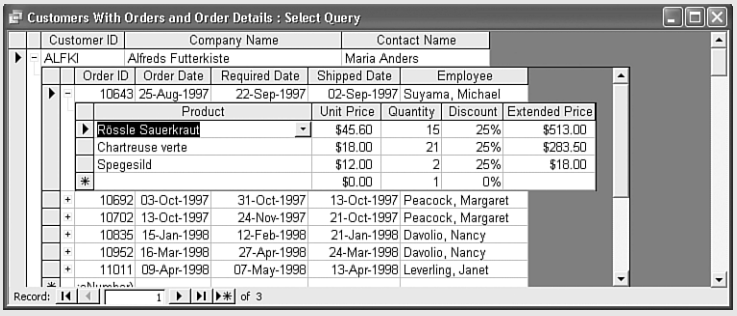


Figure 3.29 shows the resulting Customers query with displayed subdatasheets for the Orders With Extended Order Details query and the Order Details With Extended Prices query.

Figure 3.29
Drilling down from the Customers-based query to the Orders With Extended Order Details query and to the Order Details With Extended Prices query.



From Here

- For more about a query's Unique Values property, **see** "Creating a Unique Values Query," **p. 86.**
- For the details on creating multiple-table queries using SQL, **see** "Using SQL with Multiple-Table Queries," **p. 153.**
- To learn how to use multiple tables in a form, **see** "Creating a Multiple-Table Form," **p. 249.**
- To learn how to use multiple tables in a report, **see** "Creating a Multiple-Table Report," **p. 341**.

Creating Advanced Queries

4

IN THIS CHAPTER

The query techniques you studied in the first three chapters of this book add powerful new weapons to your Access arsenal. From expressions to functions to multiple-table setups, you've seen that you can perform some sophisticated and powerful querying. So when I tell you that this chapter covers "advanced" queries, am I talking about obscure features that will be rarely used in a business environment? Are these complex techniques that require a math degree to comprehend? The answer is a firm "No" on both counts. The queries you'll see in this chapter are "advanced" only in the sense that they'll advance your knowledge of how to get the most out of Access querying for your business needs.

Creating a Totals Query

A totals query includes a column that performs an aggregate operation—such as summing or averaging—on the values of a particular field. A totals query derives either a single value for the entire dynaset or several values for the records that have been grouped within in the dynaset. Table 4.1 outlines the aggregate operations you can use for your totals queries.