Before you begin creating reports, it is important that you have a good understanding of relational databases. In the first part of this chapter, you will see the tables, rows, and columns that make up relational databases. You also learn about concepts such as normalization and relationships. These are the characteristics that make a relational database . . ., well, relational.

Once you cover the basics, you are introduced to Galactic Delivery Services (GDS). The business needs of GDS serve as the basis for all the sample reports throughout this book. Even though GDS is a unique company in many respects, you will discover its reporting needs and its uses of Reporting Services are typical of most companies in this galaxy.

For the remainder of the chapter, you explore the ins and outs of the *SELECT query*, which is what you use to extract data from your data sources for use in your reports. Even though Reporting Services helps you create SELECT queries through a tool called the Query Designer, it is important that you understand how SELECT queries work and how they can be used to obtain the correct data. A report may look absolutely stunning with charts, graphics, special formatting, and snappy colors, but it is completely useless if it contains the wrong data!

Database Structure

Databases are basically giant containers for storing information. They are the electronic crawlspaces and digital attics of the corporate, academic, and governmental worlds. For example, anything that needs to be saved for later use by payroll, inventory management, or the external auditor is placed in a database.

Just like our crawlspaces and attics at home, the information placed in a database needs to be organized and classified. Figure 3-1 shows my attic in its current state. As you can see, it is going to be pretty hard to find those old kids' clothes for the thrift store clothing drive! I know they are up there somewhere.

Without some type of order placed on it, all the stuff in our home storage spaces becomes impossible to retrieve when we need it. The same is true in the world of electronic storage, as shown in Figure 3-2. Databases, like attics, need structure. Otherwise, we won't be able to find anything!

Getting Organized

The first step in getting [illegible] for everything and to have everything in its place. [illegible] ucture to the storage space, whether this [illegible] a space for data storage like a database. [illegible] have discipline of one

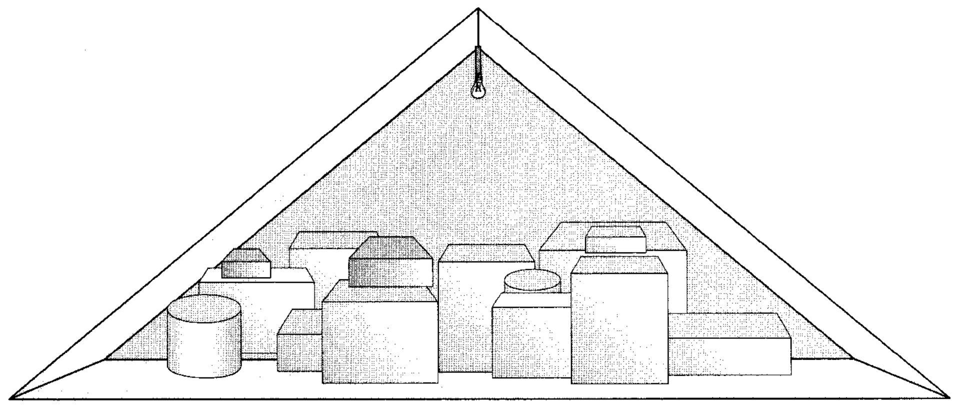
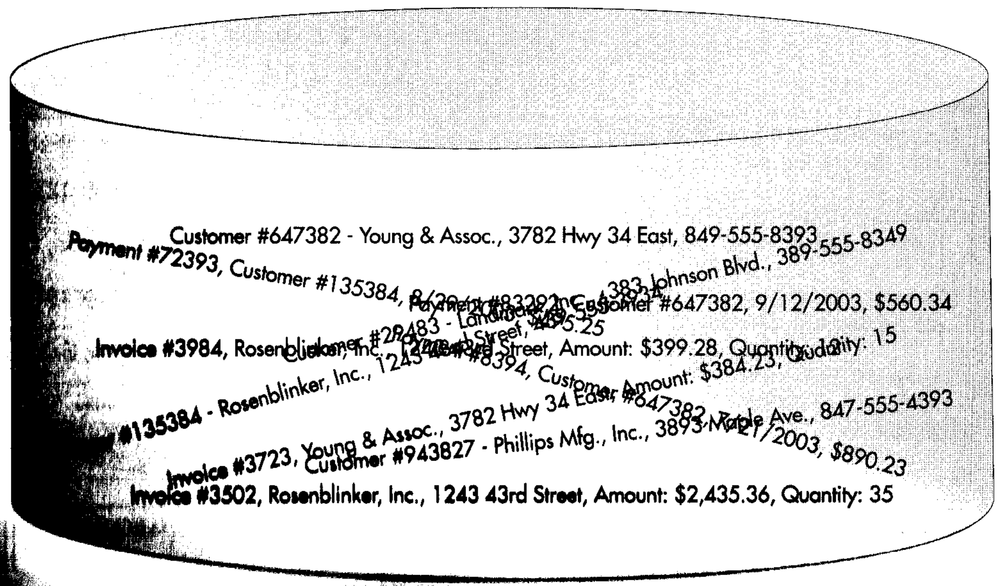


Figure 3-1 *My attic, with no organization*

Tables, Rows, and Columns

To get my attic organized, I need some shelves, a few labels, and some free time, so I can add the much-needed structure to this storage space. To keep my attic organized, I also need the discipline to pay attention to my new signs each time I put another box into storage. Figure 3-3 shows my attic as it exists in my fantasy world, where I have tons of free time and loads of self-discipline.


Customer #647382 - Young & Assoc., 3782 Hwy 34 East, 849-555-8393
Payment #72393, Customer #135384,
383 Johnson Blvd., 389-555-8349
Customer #647382, 9/12/2003, $560.34
Invoice #3984, Rosenblinker, Inc.
Street, Amount: $399.28, Quantity: 15
Amount: $384.23
#135384 - Rosenblinker, Inc., 1243
Invoice #3723, Young & Assoc., 3782 Hwy 34 East
Customer #943827 - Phillips Mfg., Inc., 3893 Maple Ave., 847-555-4393
/2003, $890.23
Invoice #3502, Rosenblinker, Inc., 1243 43rd Street, Amount: $2,435.36, Quantity: 35
[illegible] 3-2 *An unorganized [illegible]*

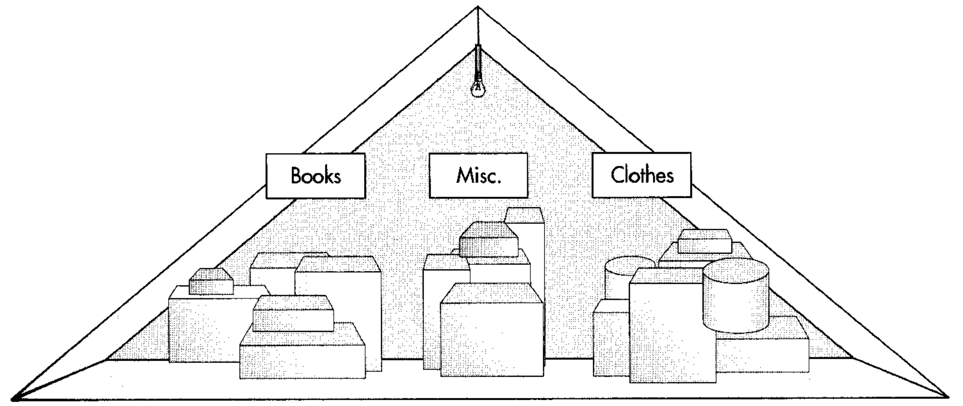


Figure 3-3 *My attic in my fantasy world*

Structure in the database world comes in the form of tables. Each database is divided into a number of tables. These tables store the information. Each table contains only one type of information. Figure 3-4 shows customer information in one table, payment information in another, and invoice header information in a third.

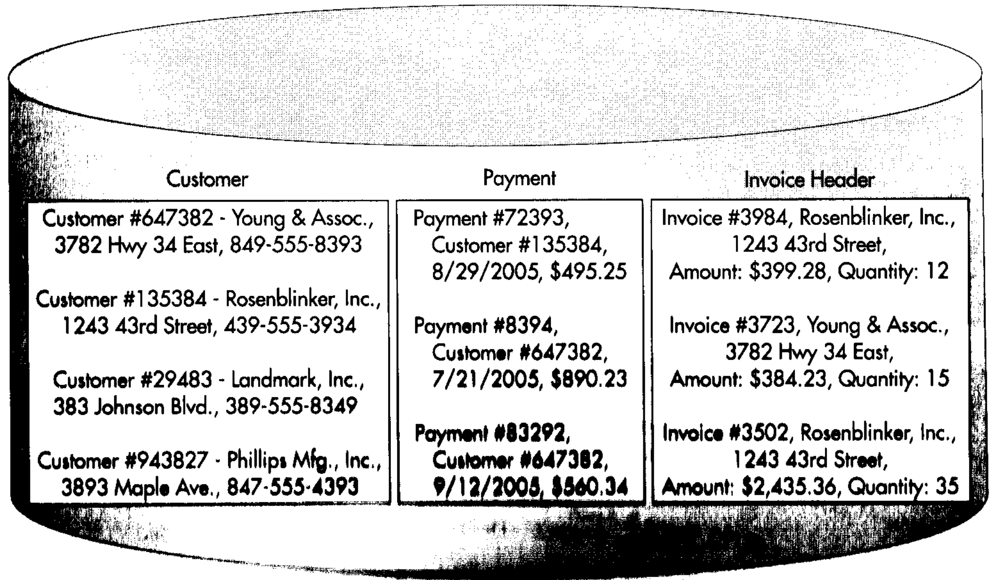

Customer
Customer #647382 - Young & Assoc., 3782 Hwy 34 East, 849-555-8393
Customer #135384 - Rosenblinker, Inc., 1243 43rd Street, 439-555-3934
Customer #29483 - Landmark, Inc., 383 Johnson Blvd., 389-555-8349
Customer #943827 - Phillips Mfg., Inc., 3893 Maple Ave., 847-555-4393
Payment
Payment #72393, Customer #135384, 8/29/2005, $495.25
Payment #8394, Customer #647382, 7/21/2005, $890.23
Payment #83292, Customer #647382, 9/12/2005, $560.34
Invoice Header
Invoice #3984, Rosenblinker, Inc., 1243 43rd Street, Amount: $399.28, Quantity: 12
Invoice #3723, Young & Assoc., 3782 Hwy 34 East, Amount: $384.23, Quantity: 15
Invoice #3502, Rosenblinker, Inc., 1243 43rd Street, Amount: $2,435.36, Quantity: 35

Figure 3-4 *A database organ* [illegible]

NOTE

Invoice Header is used as the name of the third table for consistency with the sample database that will be introduced in the section "Galactic Delivery Services" and used throughout the remainder of the book. The Invoice Header name helps to differentiate this table from the Invoice Detail table that stores the detail lines of the invoice. The Invoice Detail table is not discussed here, but it will be present in the sample database.

Dividing each table into rows and columns brings additional structure to the database. Figure 3-5 shows the Customer table divided into several rows—one row for each customer whose information is being stored in the table. In addition, the Customer table is divided into a number of columns. Each column is given a name: Customer Number, Customer Name, Address, and Phone. These names tell you what information is being stored in each column.

With a database structured as tables, rows, and columns, you know exactly where to find a certain piece of information. For example, it is pretty obvious that the customer name for customer number 135384 will be found in the Name column of the second row of the Customer table. We are starting to get this data organized, and it was a lot easier than cleaning out the attic!

NOTE

Rows in a database are also called records. *Columns in a database are also called* fields. *Reporting Services uses the terms "rows" and "records" interchangeably. It also uses the terms "columns" and "fields" interchangeably. Don't be confused by this!*

Customer

Customer #	Customer Name	Address	Phone
647382	Young & Assoc.	3782 Hwy 34 East	849-555-8393
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934
29483	Landmark, Inc.	383 Johnson Blvd.	389-555-8349
943827	Phillips Mfg., Inc.	3893 Maple Ave.	847-555-4393

Figure 3-5 *A database table* [illegible] *rows and columns*

Columns also force some discipline on anyone putting data into the table. Each column has certain characteristics assigned to it. For instance, the Customer Number column in Figure 3-5 may only contain strings of digits (0–9); no letters (A–Z) are allowed. It is also limited to a maximum of six characters. In data design lingo, these are known as *constraints*. Given these constraints, it is impossible to store a customer's name in the Customer Number column. The customer's name is likely too long and contains characters that are not legal in the Customer Number column. Constraints provide the discipline to force organization within a database.

Typically, when you design a database, you create tables for each of the things you want to keep track of. In Figure 3-4, the database designer knew that her company needed to track information for customers, payments, and invoices. Database designers call these things *entities*. The database designer created tables for the customer, payment, and invoice header entities. These tables are named Customer, Payment, and Invoice Header.

Once the entities have been identified, the database designer determines what information needs to be known about each entity. In Figure 3-5, the designer identified the customer number, customer name, address, and phone number as the things that need to be known for each customer. These are *attributes* of the customer entity. The database designer creates a column in the Customer table for each of these attributes.

Primary Key

As entities and attributes are being defined, the database designer needs to identify a special attribute for each entity in the database. This special attribute is known as the primary key. The purpose of the *primary key* is to be able to uniquely identify a single entity or, in the case of a database table, a single row in the table.

Two simple rules exist for primary keys. First, every entity must have a primary key value. Second, no two rows in an entity can have the same primary key value. In Figure 3-5, the Customer Number column can serve as the primary key. Every customer is assigned a customer number and no two customers can be assigned the same customer number.

For most entities, the primary key is a single attribute. However, in some cases, two attributes must be combined to create a unique primary key. This is known as a *composite primary key*. For instance, if you were defining an entity based on Presidents of the United States, the first name would not be a valid primary key. John Adams, John Quincy Adams, and John Kennedy all have the same first name. You would need to create a composite key combining first name, middle name, and last name to have a valid primary key.

Normalization

As the database designer continues to work on identifying entities and attributes, she will notice that two different entities have some of the same attributes. For example, in Figure 3-6, both the customer entity and the invoice header entity have attributes of Customer Name and Address. This duplication of information seems rather wasteful. Not only are the customer's name and address duplicated between the Customer and Invoice Header tables, but they are also duplicated in several rows in the Invoice Header table itself.

Customer

Customer #	Customer Name	Address	Phone
647382	Young & Assoc.	3782 Hwy 34 East	849-555-8393
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934
29483	Landmark, Inc.	383 Johnson Blvd.	389-555-8349
943827	Phillips Mfg., Inc.	3893 Maple Ave.	847-555-4393

Invoice Header

Invoice #	Customer Name	Address	Amount	Quantity
3984	Rosenblinker, Inc.	1243 43rd Street	$399.28	12
3723	Young & Assoc.	3782 Hwy 34 East	$384.23	15
3502	Rosenblinker, Inc.	1243 43rd Street	$2,435.36	35

Database tables with duplicate data

The duplicate data also leads to another problem. Suppose that Rosenblinker, Inc. changes its name to RB, Inc. Then, Ann in the data-processing department changes the name in the Customer table because this is where we store information about the customer entity. However, the customer name has not been changed in the Invoice Header table. Because the customer name in the Invoice Header table no longer matches the customer name in the Customer table, it is no longer possible to determine how many invoices are outstanding for RB, Inc. Believe me, the accounting department will think this is a bad situation.

To avoid these types of problems, database tables are normalized. *Normalization* is a set of rules for defining database tables, so each table contains attributes from only one entity. The rules for creating normalized database tables can be quite complex. You can hear database designers endlessly debating whether a proper database should be in third normal form, fourth normal form, or one hundred and twenty-seventh normal form. Let the database designers debate all they want. All you need to remember is this: a normalized database avoids data duplication.

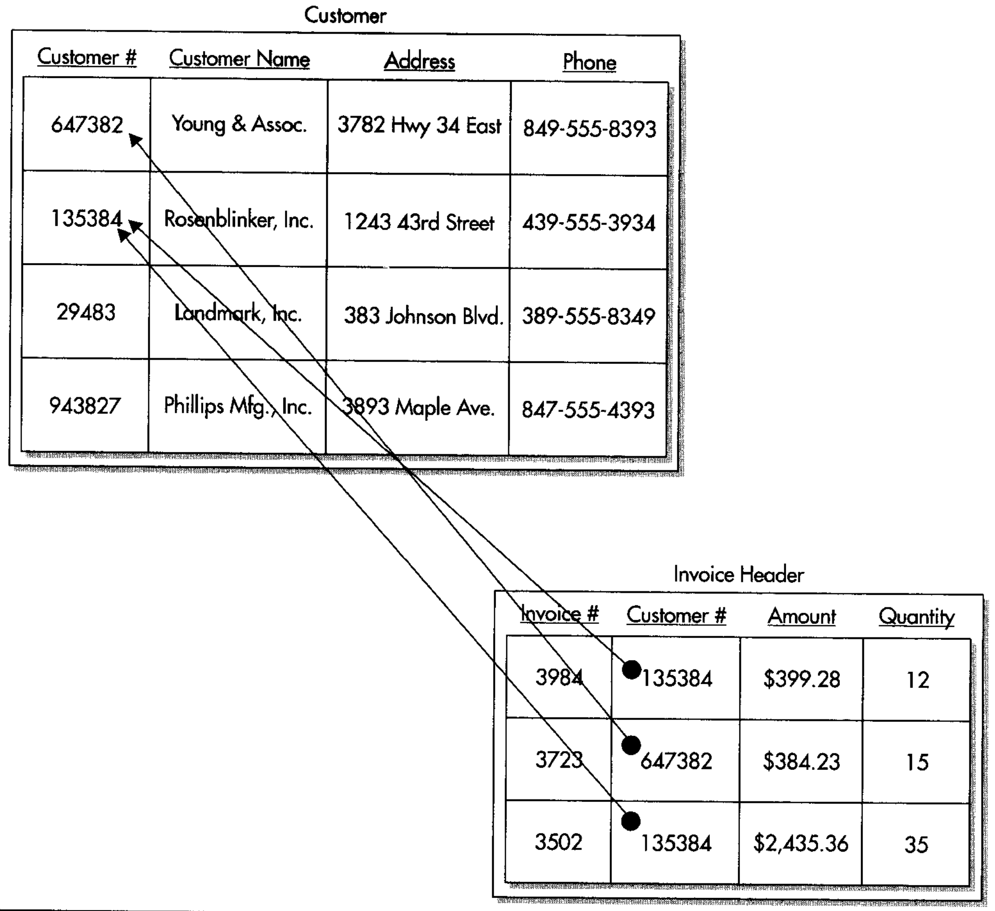
Relations

A *relation* is a tool the database designer uses to avoid data duplication when creating a normalized database. A relation is simply a way to put the duplicated data in one place, and then point to it from all the other places in the database where it would otherwise occur. The table that contains the data is called the *parent* table. The table that contains a pointer to the data in the parent table is called the *child* table. Just like parents and children of the human variety, the parent table and the child table are said to be *related.*

In our example, the customer name and address are stored in the Customer table. This is the parent table. A pointer is placed in the Invoice Header table in place of the duplicate customer names and addresses it had contained. The Invoice Header table is the child table.

As mentioned previously, each customer is uniquely identified by their customer number. Therefore, the Customer Number column serves as the primary key for the Customer table. In the Invoice Header table, we need a way to point to a particular customer. It makes sense to use the primary key in the parent table, in this case the Customer Number column, as that pointer. This is illustrated in Figure 3-7.

Each row in the Invoice Header table now contains a copy of the primary key of a row in the Customer table. The Customer Number column in the Invoice Header table is called the foreign key. It is called a *foreign key* because it is not one of the native attributes of the invoice header entity. The customer number is a native attribute of the customer entity. The only reason the Customer Number column exists in the Invoice Header table is to create the relationship.


Customer

Customer #	Customer Name	Address	Phone
647382	Young & Assoc.	3782 Hwy 34 East	849-555-8393
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934
29483	Landmark, Inc.	383 Johnson Blvd.	389-555-8349
943827	Phillips Mfg., Inc.	3893 Maple Ave.	847-555-4393

Invoice Header

Invoice #	Customer #	Amount	Quantity
3984	135384	$399.28	12
3723	647382	$384.23	15
3502	135384	$2,435.36	35

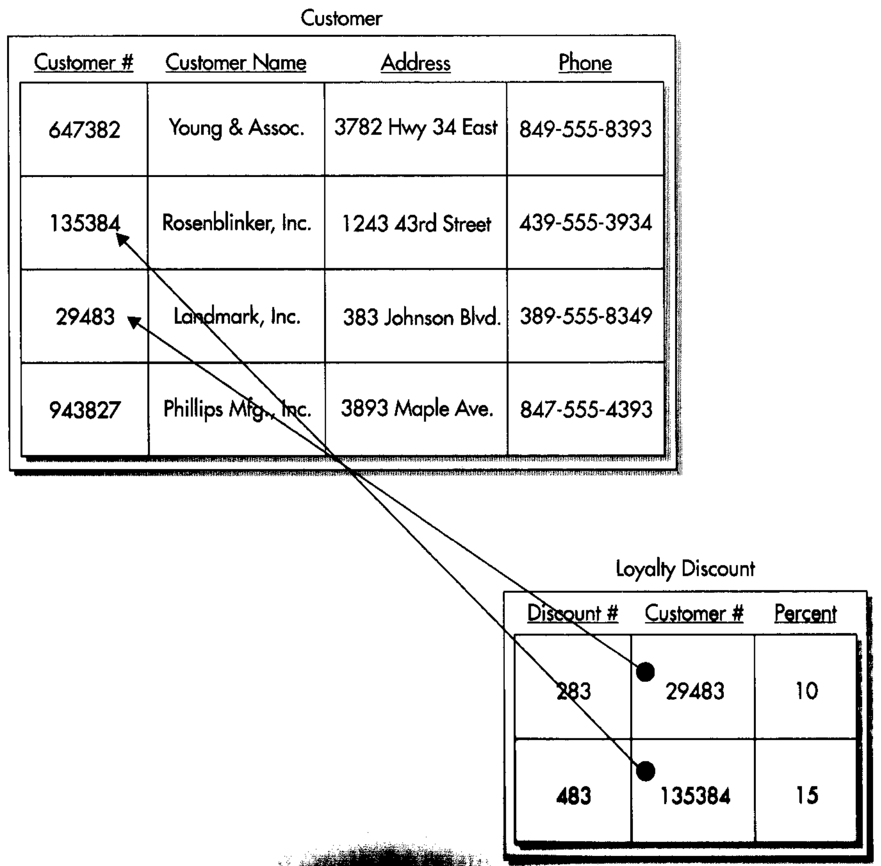
Figure 3-7 *A database relation*

Let's look back at the name change problem, this time using our new database structure that includes the parent-child relationship. When Rosenblinker, Inc. changes its name to RB, Inc., Ann changes the name in the Customer table as before. In our new structure, however, the customer name is not stored in any other location. Instead, the Invoice Header table rows for RB, Inc. point back to the Customer table row that has the correct name. The accounting department stays happy because it can still figure out how many invoices are outstanding for RB, Inc.

Cardinality of Relations

Database relations can be classified by the number of records that can exist on each side of the relationship. This is known as the *cardinality* of the relation. For example, the relation in Figure 3-7 is a *one-to-many relation* (in other words, one parent record can have many children). More specifically, one customer can have many invoices.

It is also possible to have a *one-to-one relation*. In this case, one parent record can have only one child. For example, let's say our company rewards customers with a customer loyalty discount. Because only a few customers will receive this loyalty discount, we do not want to set aside space in every row in the Customer table to store the loyalty discount information. Instead, we create a new table to store this information. The new table is related to the Customer table, as shown in Figure 3-8.



Customer

Customer #	Customer Name	Address	Phone
647382	Young & Assoc.	3782 Hwy 34 East	849-555-8393
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934
29483	Landmark, Inc.	383 Johnson Blvd.	389-555-8349
943827	Phillips Mfg., Inc.	3893 Maple Ave.	847-555-4393

Loyalty Discount

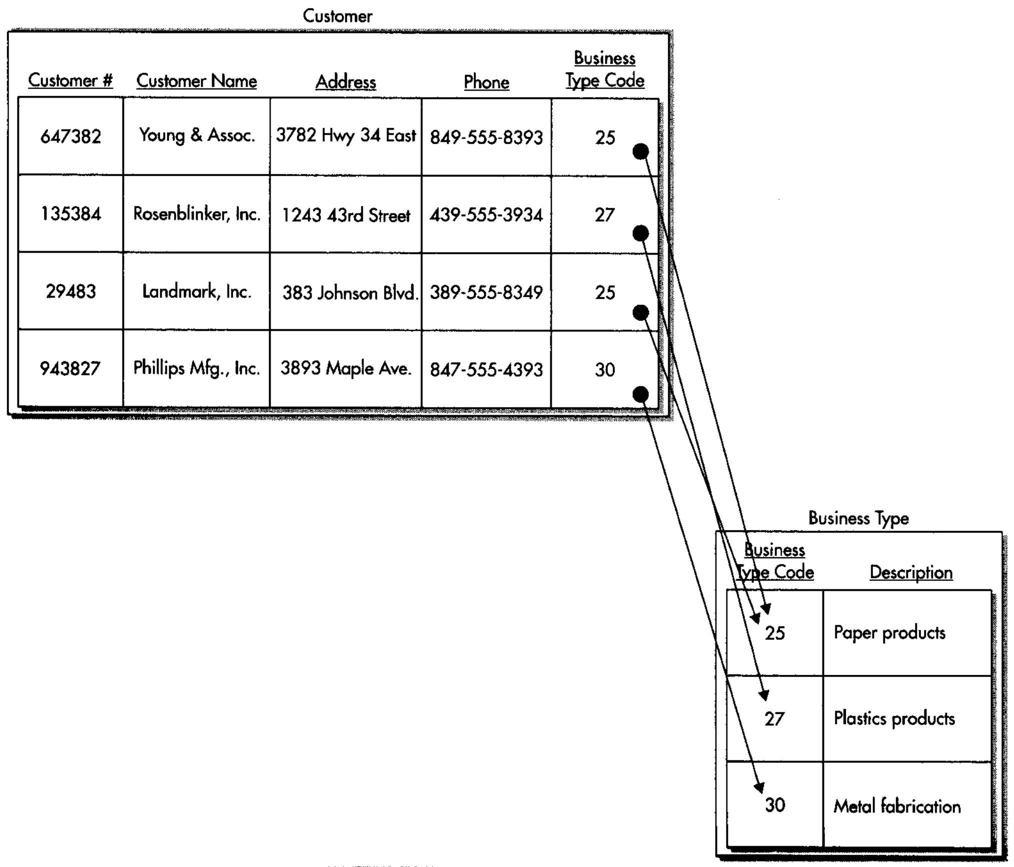
Discount #	Customer #	Percent
283	29483	10
483	135384	15

Figure 3-8 ***A one-to-one [illegible]***

Our company's business rule says that a given customer can only receive one loyalty discount. Because the Loyalty Discount table has only one Customer Number column, each row can link to just one customer. The combination of the business rule and the table design make this a one-to-one relation.

It is also possible to have a *many-to-many relation*. This relation no longer fits our parent/child analogy. It is better thought of as a brother/sister relationship. One brother can have many sisters, and one sister can have many brothers.

Suppose we need to keep track of the type of business engaged in by each of our customers. We can add a Business Type table to our database with columns for the business type code and the business type description. We can add a column for the business type code to the Customer table. We now have a one-to-many relation, where one business type can be related to many customers. This is shown in Figure 3-9.



Customer

Customer #	Customer Name	Address	Phone	Business Type Code
647382	Young & Assoc.	3782 Hwy 34 East	849-555-8393	25
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934	27
29483	Landmark, Inc.	383 Johnson Blvd.	389-555-8349	25
943827	Phillips Mfg., Inc.	3893 Maple Ave.	847-555-4393	30

Business Type

Business Type Code	Description
25	Paper products
27	Plastics products
30	Metal fabrication

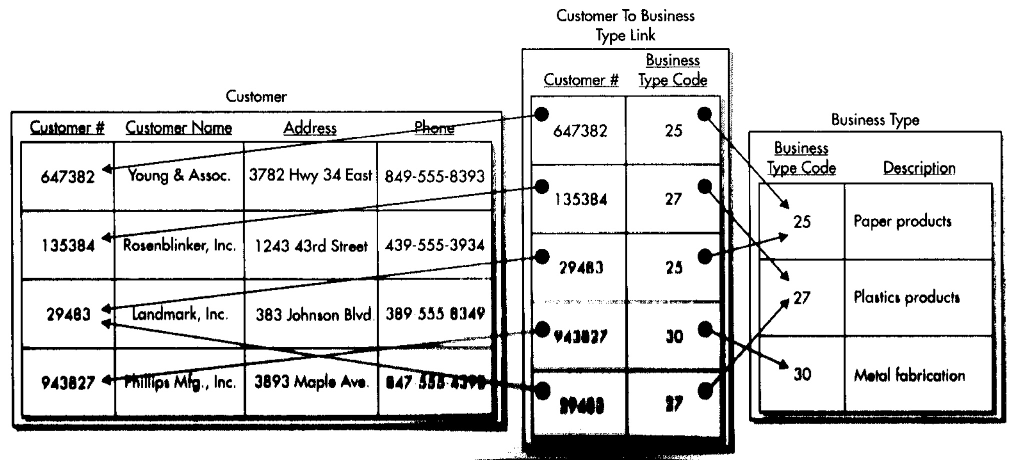
Figure 3-9 ***Tracking business type using a one-to-many relation***

The problem with this structure becomes apparent when we have a customer that does multiple things. If Landmark, Inc. only produces paper products, there isn't a problem. We can put the business type code for paper products in the Customer table row for Landmark, Inc. We run into a bit of a snag, however, if Landmark, Inc. also produces plastics. We could add a second business type code column to the Customer table, but this still limits a customer to a maximum of two business types. In today's world of national conglomerates, this is not going to work.

The answer is to add a third table to the mix to create a many-to-many relationship. This additional table is known as a *linking table*. Its only purpose is to link two other tables together in a many-to-many relation. To use a linking table, you create the Business Type table just as before. This time, instead of creating a new column in the Customer table, we'll create a new table called Customer To Business Type Link. The new table has columns for the customer number and the business type code. Figure 3-10 shows how this linking table relates the Customer table to the Business Type table. By using the linking table, we can relate one customer to many business types. In addition, we can relate one business type to many customers.

Retrieving Data

We now have all the tools we need to store our data in an efficient manner. With our data structure set, it is time to determine how we can access that data to use it in our reports. Data that was split into multiple tables must be recombined for reporting. This is done using a database tool called a *join*. In most cases, we will also want the data in the report to appear in a certain order. This is accomplished using a sort.



Customer #	Customer Name	Address	Phone
647382	Young & Assoc.	3782 Hwy 34 East	849-555-8393
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934
29483	Landmark, Inc.	383 Johnson Blvd.	389 555 8349
943827	Phillips Mfg., Inc.	3893 Maple Ave.	847 555 [illegible]

Customer #	Business Type Code
647382	25
135384	27
29483	25
943827	30
29483	27

Business Type Code	Description
25	Paper products
27	Plastics products
30	Metal fabrication

Figure 3-10 Tracking the business type u[illegible] [illegible]ny relation

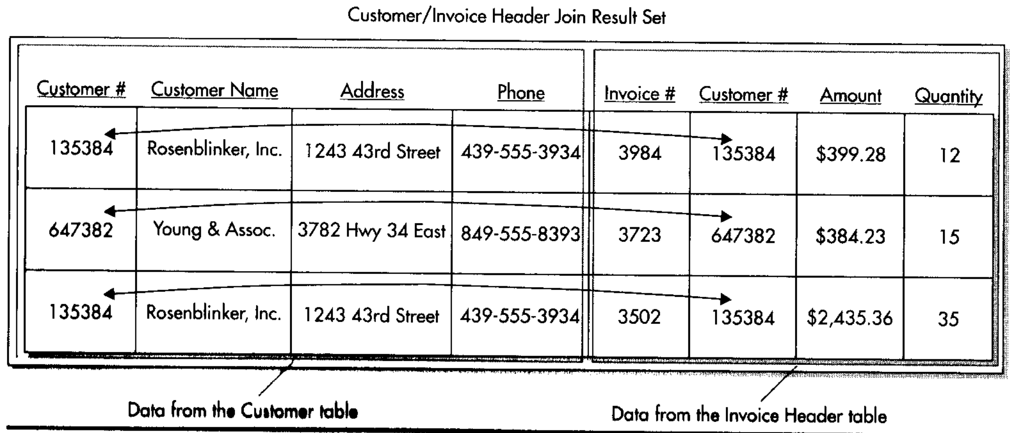
Inner Joins

Suppose we need to know the name and address of the customer associated with each invoice. This is certainly a reasonable request, especially if we want to send invoices to these clients and have those invoices paid. Checking the Invoice Header table, you can see it contains the customer number, but not the name and address. The name and address is stored in the Customer table.

To print our invoices, we need to join the data in the Customer table with the data in the Invoice Header table. This join is done by matching the customer number in each record of the Invoice Header table with the customer number in the Customer table. In the language of database designers, we are joining the Customer table to the Invoice Header table on the Customer Number column.

The result of the join is a new table that contains information from both the Customer table and the Invoice Header table in each row. This new table is known as a *result set*. The result set from the Customer table–to–Invoice Header table join is shown in Figure 3-11. Note, the result set table contains nearly the same information that was in the Invoice Header table before it was normalized. The result set is a *denormalized* form of the data in the database.

It may seem like we are going in circles, first normalizing the data, and then denormalizing it. There is, however, one important difference between the denormalized form of the Invoice Header table that we started with in Figure 3-6 and the result set in Figure 3-11. The denormalized result set is a temporary table: it exists only as long as it is needed; then it is automatically deleted. The result set is re-created each time we execute the join, so the result set is always current.



Customer/Invoice Header Join Result Set
Data from the Customer table
Data from the Invoice Header table

Customer #	Customer Name	Address	Phone	Invoice #	Customer #	Amount	Quantity
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934	3984	135384	$399.28	12
647382	Young & Assoc.	3782 Hwy 34 East	849-555-8393	3723	647382	$384.23	15
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934	3502	135384	$2,435.36	35

Figure 3-11 The result set from the Customer table–to–Invoice Header table join

Let's return once more to Ann, our faithful employee in Data Processing. We will again consider the situation where Rosenblinker, Inc. changes its name to RB, Inc. Ann makes the change in the Customer table, as in the previous example. The next time we execute the join, this change is reflected in the result set. The result set has the new company name because our join gets a new copy of the customer information from the Customer table each time it is executed. The join finds the information in the Customer table based on the primary key, the customer number, which has not changed. Our invoices are linked to the proper companies, so Accounting can determine how many invoices are outstanding for RB, Inc., and everyone is happy!

Outer Joins

In the previous section, we looked at a type of join known as an inner join. When you do an inner join, your result set includes only those records that have a representative on both sides of the join. In Figure 3-11, Landmark, Inc. and Phillips Mfg., Inc. are not represented in the result set, because they do not have any Invoice Header table rows linked to them.

Figure 3-12 shows another way to think about joins. Here, the two tables are shown as sets of customer numbers. The left-hand circle represents the set of customer numbers in the Customer table. It contains one occurrence of every customer number present in the Customer table. The right-hand circle represents the set of customer numbers in the

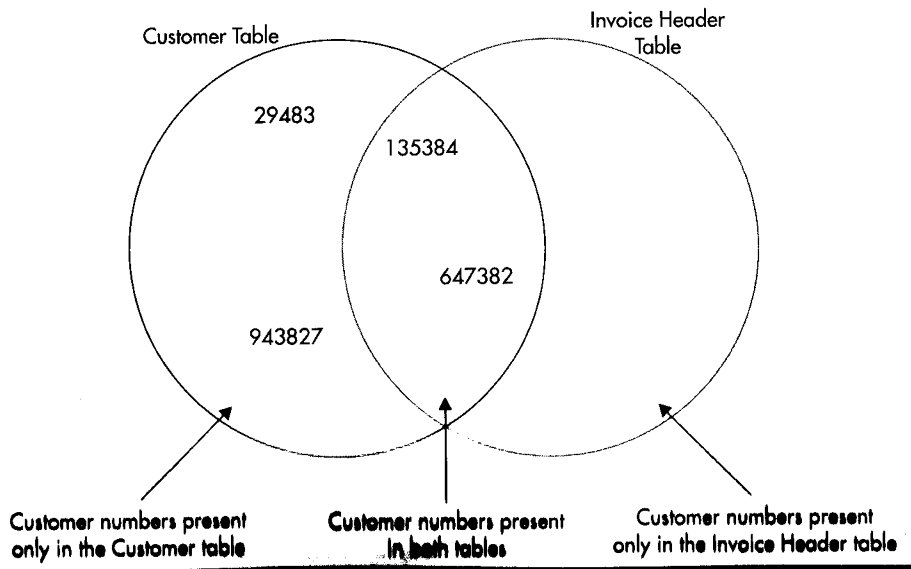

Customer Table
Invoice Header Table
29483
135384
647382
943827
Customer numbers present only in the Customer table
Customer numbers present in both tables
Customer numbers present only in the Invoice Header table

Figure 3-12 ***The set repres[illegible] Customer and Invoice Header tables***

Invoice Header table. It contains one occurrence of each customer number present in the Invoice Header table. The center region, where the two sets intersect, contains one occurrence of every customer number present in both the Customer table and the Invoice Header table. Looking at Figure 3-12, you can quickly tell no customer numbers are present in the Invoice Header table, but not in the Customer table. This is as it should be. We should not have any invoice headers assigned to a customer that do not exist in the Customer table.

Figure 3-13 shows a graphical representation of the inner join in Figure 3-11. Only records with customer numbers that appear in the shaded section will be included in the result set. Remember, two rows in the Invoice Header table contain customer number 135384. For this reason, the result set contains three rows—two rows for customer number 135384 and one row for customer number 647382.

The result set in Figure 3-11 enables us to print invoice headers that contain the correct customer name and address. Now let's look at customers and invoice headers from a slightly different angle. Suppose we have been asked for a report showing all customers and the invoice headers that have been sent to them. If we were to print this customers/invoice headers report from the result set in Figure 3-11, it would exclude Landmark, Inc. and Phillips Mfg., Inc. because they do not have any invoices and, therefore, would not fulfill the requirements.

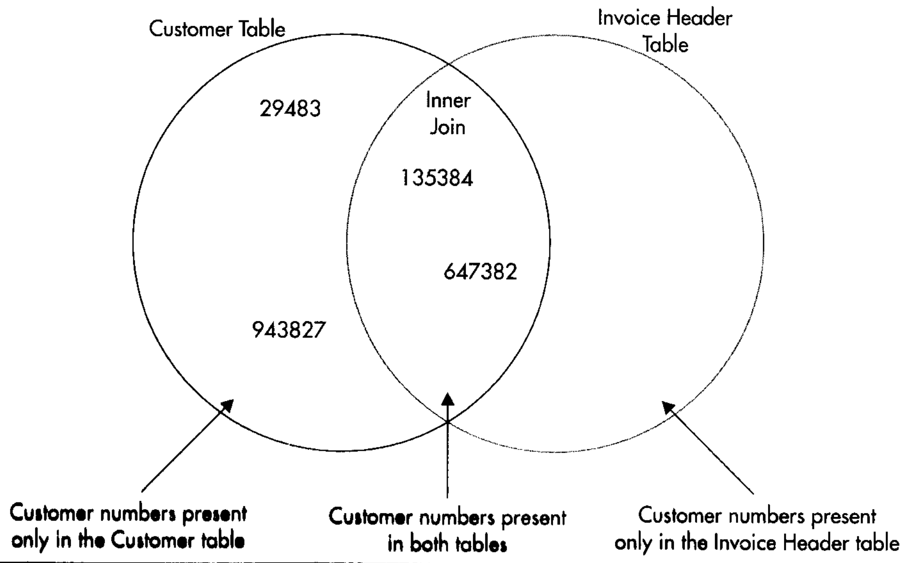

Customer Table
Invoice Header Table
29483
Inner Join
135384
647382
943827
Customer numbers present only in the Customer table
Customer numbers present in both tables
Customer numbers present only in the Invoice Header table

Figure 3-13 ***The set representation of the inner join of the Customer table and the Invoice Header table***

What we need is a result set that includes all the customers in the Customer table. This is illustrated graphically in Figure 3-14. This type of join is known as a *left outer join*, so named because this join is not limited to the values in the intersection of both circles. It also includes the values to the left of the inner, overlapping sections of the circles.

We can also perform a right outer join on two tables. In our example, a *right outer join* would return the same number of rows as the inner join. This is because no customer numbers are to the right of the intersection.

The result set produced by a left outer join of the Customer table and the Invoice Header table is shown in Figure 3-15. Notice the columns populated by data from the Invoice Header table are empty in rows for Landmark, Inc. and Phillips Mfg., Inc. The columns are empty because these two customers do not have any Invoice Header rows to provide data on the right side of the join.

Joining Multiple Tables

Joins, whether inner or outer, always involve two tables. However, in Figure 3-10, you were introduced to a many-to-many relation that involved three tables. How do you retrieve data from this type of relation? The answer is to chain together two different joins, each involving two tables.

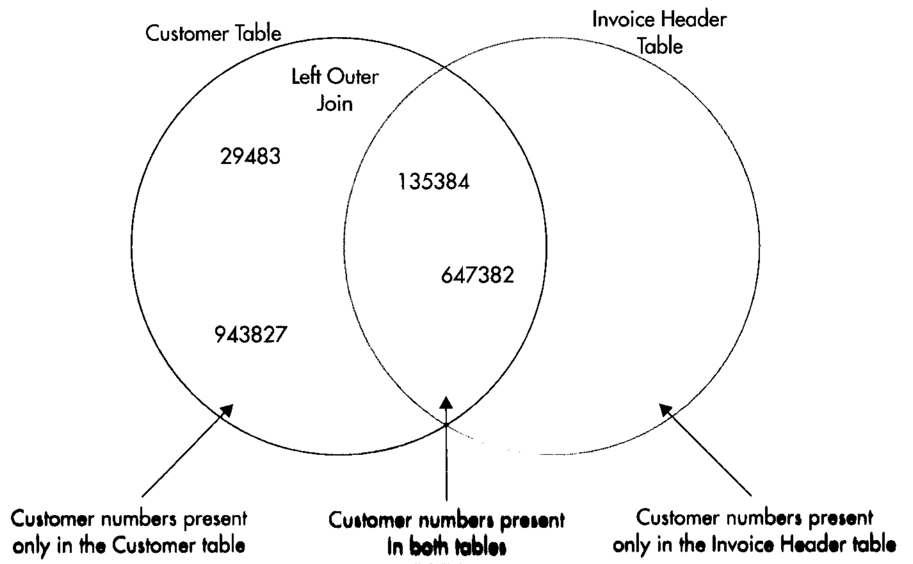
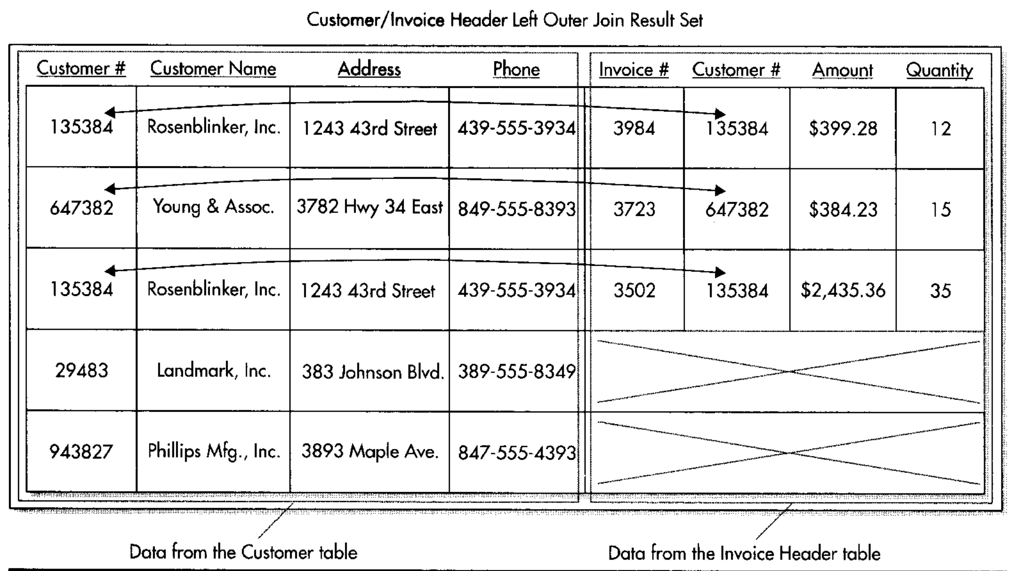

Customer Table
Invoice Header Table
Left Outer Join
29483
943827
135384
647382
Customer numbers present only in the Customer table
Customer numbers present in both tables
Customer numbers present only in the Invoice Header table

Figure 3-14 *The set representation of the left outer join of the Customer table and the Invoice Header table*


Customer/Invoice Header Left Outer Join Result Set

Customer #	Customer Name	Address	Phone	Invoice #	Customer #	Amount	Quantity
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934	3984	135384	$399.28	12
647382	Young & Assoc.	3782 Hwy 34 East	849-555-8393	3723	647382	$384.23	15
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934	3502	135384	$2,435.36	35
29483	Landmark, Inc.	383 Johnson Blvd.	389-555-8349				
943827	Phillips Mfg., Inc.	3893 Maple Ave.	847-555-4393				

Data from the Customer table
Data from the Invoice Header table

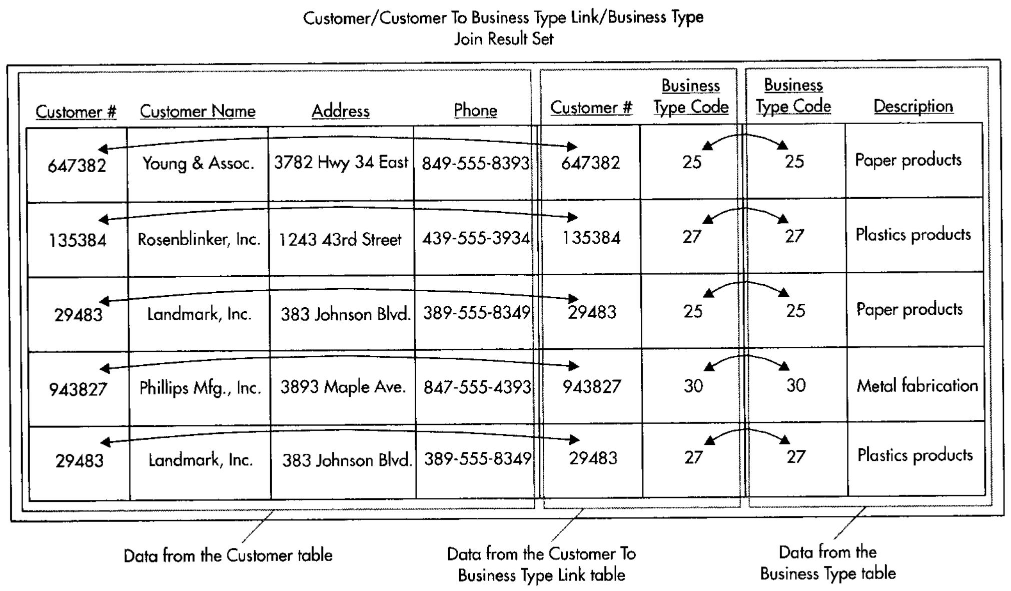
Figure 3-15 *The result set from the left outer join of the Customer table and the Invoice Header table*

Figure 3-16 illustrates the joins required to reassemble the data from Figure 3-10. Here, the Customer table is joined to the Customer To Business Type Link table using the Customer Number column common to both tables. The Customer To Business Type Link table is then joined to the Business Type table using the Business Type Code column present in both tables. The final result set contains the data from all three tables.

Self-Joins

In our previous example, we needed to join three tables to get the required information. Other joins may only require a single table. For instance, we may have a customer that is a subsidiary of another one of our customers. In some cases, we'll want to treat these two separately, so both appear in our result set. This requires us to keep the two customers as separate rows in our Customer table. In other cases, we may want to combine information from the parent company and the subsidiary into one record. To do this, our database structure must include a mechanism to tie the subsidiary to its parent.

To track a customer's connection to its parent, we need to create a relationship between the customer's row in the Customer table and its parent's row in the Customer table. To do this, we add a Parent Customer Number column to the Customer table,



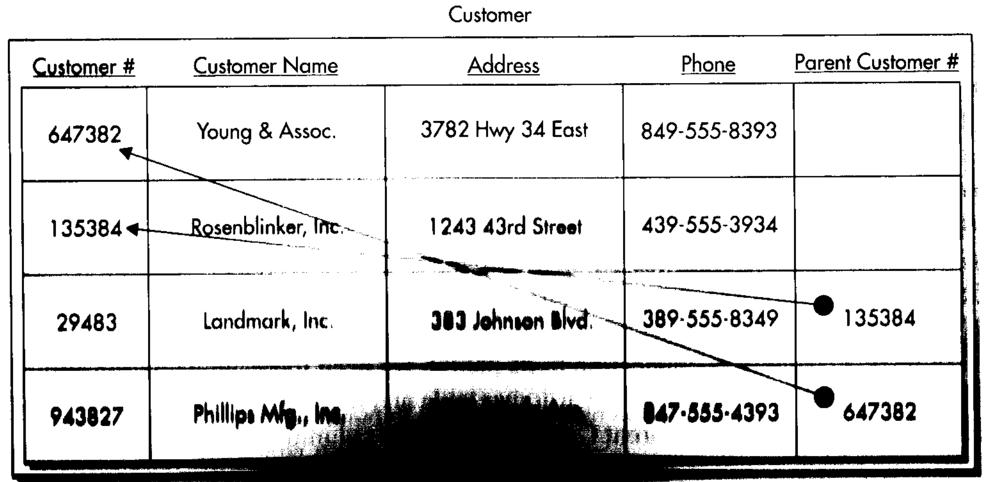
Customer/Customer To Business Type Link/Business Type Join Result Set

Customer #	Customer Name	Address	Phone	Customer #	Business Type Code	Business Type Code	Description
647382	Young & Assoc.	3782 Hwy 34 East	849-555-8393	647382	25	25	Paper products
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934	135384	27	27	Plastics products
29483	Landmark, Inc.	383 Johnson Blvd.	389-555-8349	29483	25	25	Paper products
943827	Phillips Mfg., Inc.	3893 Maple Ave.	847-555-4393	943827	30	30	Metal fabrication
29483	Landmark, Inc.	383 Johnson Blvd.	389-555-8349	29483	27	27	Plastics products

Data from the Customer table
Data from the Customer To Business Type Link table
Data from the Business Type table

Figure 3-16 *The result set from the join of the Customer table, the Customer To Business Type Link table, and the Business Type table*

as shown in Figure 3-17. In the customer's row, the Parent Customer Number column will contain the customer number of the row for the parent. In the row for the parent, and in all the rows for customers that do not have a parent, the Parent Customer Number column is empty.



Customer

Customer #	Customer Name	Address	Phone	Parent Customer #
647382	Young & Assoc.	3782 Hwy 34 East	849-555-8393	
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934	
29483	Landmark, Inc.	383 Johnson Blvd.	389-555-8349	135384
943827	Phillips Mfg., Inc.	[illegible]	847-555-4393	647382

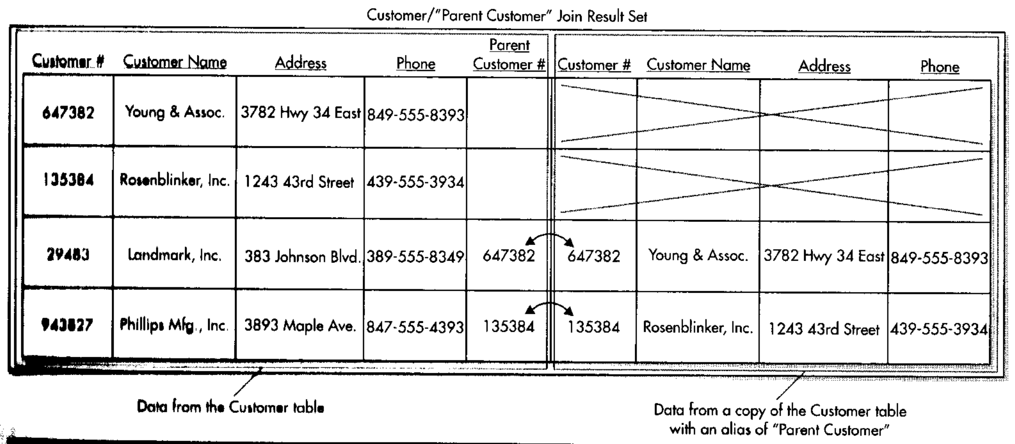
Figure 3-17 *The Cu*

When we want to report from this parent/subsidiary relation, we need to do a join. This may seem like a problem at first because a join requires two tables, and we only have one. The answer is to use the Customer table on one side of the join and a "copy" of the Customer table on the other side of the join. The second occurrence of the Customer table is given a nickname, called an *alias*, so we can tell the two apart. This type of join, which uses the same table on both sides, is known as a *self-join*. Figure 3-18 shows the results of the self-join on the Customer table.

Sorting

In most cases, one final step is required before our result sets can be used for reporting. Let's go back to the result set produced in Figure 3-15 for the customers/invoice headers report. Looking back at this result set, notice the customers do not appear to be in any particular order. In most cases, users do not appreciate reports with information presented in this unsorted manner. This is especially true when two rows for the same customer do not appear consecutively, as is the case here.

We need to sort the result set as it is being created to avoid this situation. This is done by specifying the columns that should be used for the sort. Sorting by Customer Name probably makes the most sense for the customers/invoice headers report. Columns can be sorted either in ascending order, smallest to largest (A–Z), or descending order, largest to smallest (Z–A). An ascending sort on Customer Name would be most appropriate.

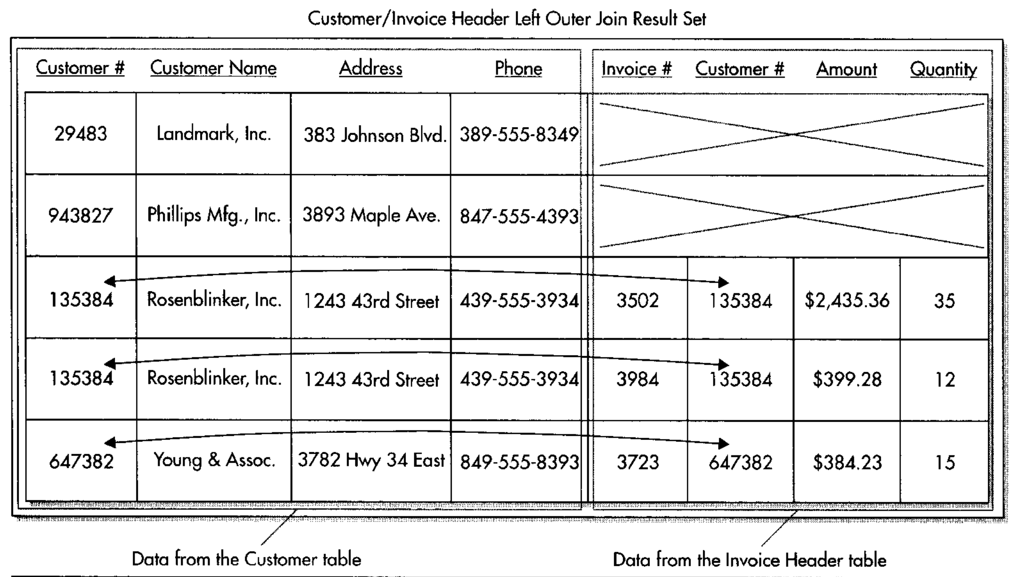


Customer/"Parent Customer" Join Result Set

Customer #	Customer Name	Address	Phone	Parent Customer #	Customer #	Customer Name	Address	Phone
647382	Young & Assoc.	3782 Hwy 34 East	849-555-8393					
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934					
29483	Landmark, Inc.	383 Johnson Blvd.	389-555-8349	647382	647382	Young & Assoc.	3782 Hwy 34 East	849-555-8393
943827	Phillips Mfg., Inc.	3893 Maple Ave.	847-555-4393	135384	135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934

Data from the Customer table
Data from a copy of the Customer table with an alias of "Parent Customer"

Figure 3-18 *The result set from the Customer table self-join*



Customer/Invoice Header Left Outer Join Result Set

Customer #	Customer Name	Address	Phone	Invoice #	Customer #	Amount	Quantity
29483	Landmark, Inc.	383 Johnson Blvd.	389-555-8349				
943827	Phillips Mfg., Inc.	3893 Maple Ave.	847-555-4393				
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934	3502	135384	$2,435.36	35
135384	Rosenblinker, Inc.	1243 43rd Street	439-555-3934	3984	135384	$399.28	12
647382	Young & Assoc.	3782 Hwy 34 East	849-555-8393	3723	647382	$384.23	15

Data from the Customer table

Data from the Invoice Header table

Figure 3-19 *The sorted result set from the left outer join of the Customer table and the Invoice Header table*

We still have a situation where the order of the rows is left to chance. Because two rows have the same customer name, we do not know which of these two rows will appear first and which will appear second. A second sort field is necessary to break this "tie." All the data copied into the result set from the Customer table will be the same in both of these rows. We need to look at the data copied from the Invoice Header table for a second sort column. In this case, an ascending sort on Invoice Number would be a good choice. Figure 3-19 shows the result set sorted by Customer Name, ascending, and then Invoice Number, ascending.

Galactic Delivery Services

Throughout the remainder of this book, you will get to know Reporting Services by exploring a number of sample reports. These reports will be based on the business needs of a company called Galactic Delivery Services (GDS). To better understand these sample reports, here is some background on GDS.

Company Background

GDS provides package-delivery service between several planetary systems in the near galactic region. It specializes in rapid delivery featuring same-day, next-day, and previous-day delivery. The latter is made possible by its new Photon III transports, which travel faster than the speed of light. This faster-than-light capability allows GDS to exploit the properties of general relativity and deliver a package on the day before it was sent.

Package Tracking

Despite GDS's unique delivery offerings, it has the same data-processing needs as any more conventional package-delivery service. It tracks packages as they are moved from one interplanetary hub to another. This is important not only for the smooth operation of the delivery service, but also to allow customers to check on the status of their delivery at any time.