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

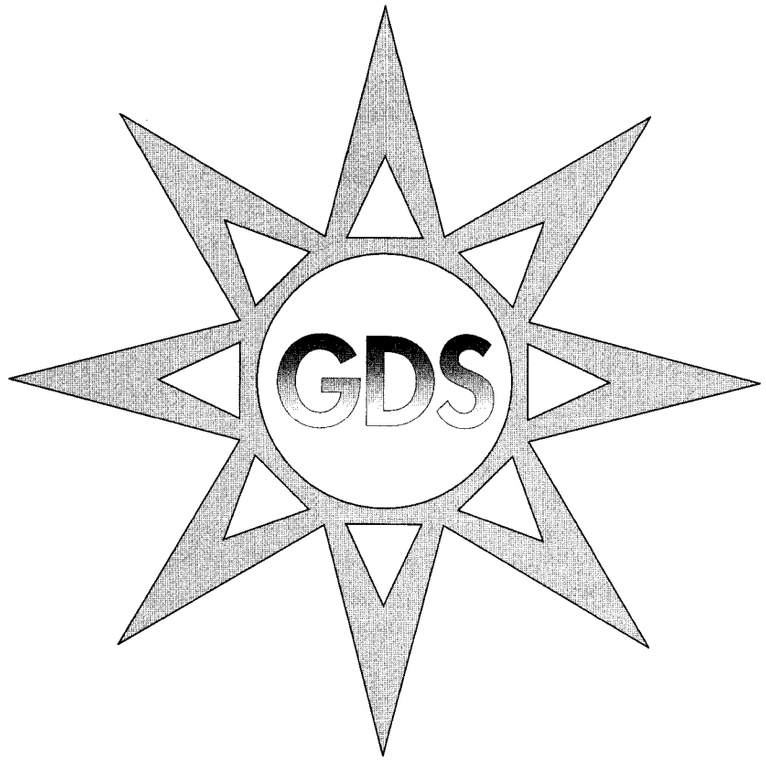
Data from the Invoice Header table

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

To remain accountable to its clients and to prevent fraud, GDS investigates every package lost en route. These investigations help to find and eliminate problems throughout the entire delivery system. One such investigation discovered that a leaking antimatter valve on one of the Photon III transports was vaporizing two or three packages on each flight.

GDS stores its data in a database called Galactic. Figure 3-20 shows the portion of the Galactic database that stores the information used for package tracking. The tables and their column names are shown. A key symbol in the gray square next to a column name indicates this column is the primary key for that table. The lines connecting the tables show the relations that have been created between these tables in the database. The key symbol at the end of the line points to the primary key column used to create the relation. The infinity sign, at the opposite end of the line to the key symbol, points to the foreign key column used to complete the relation. (The infinity sign looks like two circles or a sideways number 8.)

Each relation shown in Figure 3-20 is a one-to-many relation. The side of the relation indicated by the key is the "one" side of the relation. The side indicated by the infinity sign is the many side of the relation. For example, if you look at the line between the Customer table and the Delivery table, you can see that one customer may have many deliveries.

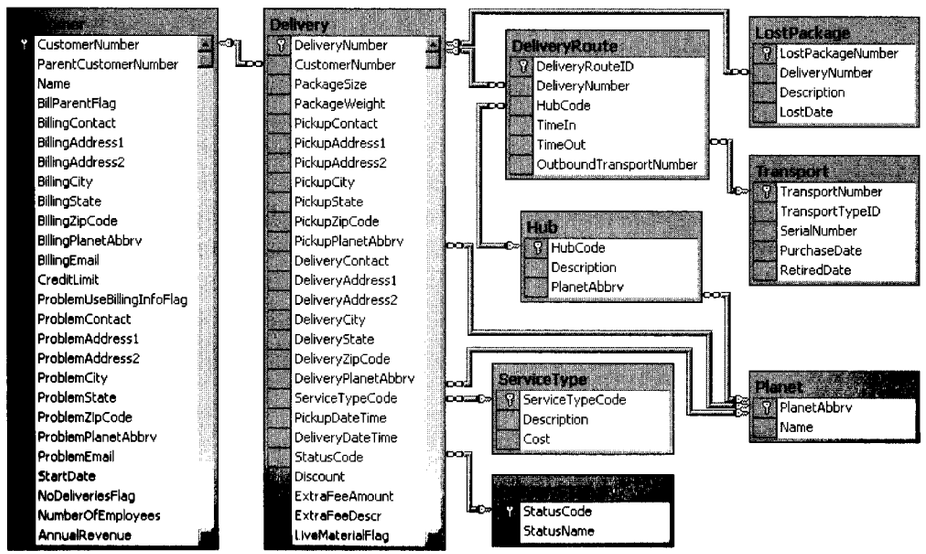


Figure 3-20 *The package tracking tables from the Galactic database*

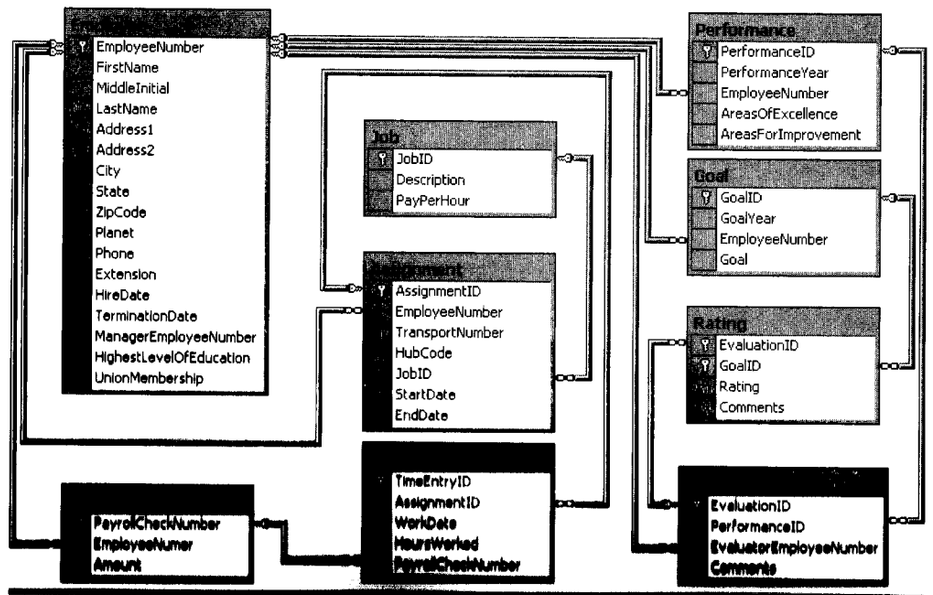
You may want to refer to these diagrams as we create sample reports from the Galactic database. Don't worry if the diagrams seem a bit complicated right now. They will make more sense as we consider the business practices and reporting needs at GDS. Also, our first report examples will contain only a few tables and the corresponding relations, so we will start simple and work our way up.

Personnel

Every business needs a personnel department to look after its employees. GDS is no different. The GDS personnel department is responsible for the hiring and firing of all the robots employed by GDS. This department is also responsible for tracking the hours put in by the robotic laborers and paying them accordingly. (Yes, robots get paid at GDS. After all, GDS is an equal-opportunity employer.)

The personnel department is also responsible for conducting annual reviews of each employee. At the annual review, goals are set for the employee to attain over the coming year. After a year has passed, several of the employee's coworkers are asked to rate the employee on how well it did in reaching those goals. The employee's manager then uses the ratings to write an overall performance evaluation for the employee and establish new goals for the following year.

Figure 3-21 shows the tables in the Galactic database used by the personnel department. Notice that the Rating table has key symbols next to both the EvaluationID



column name and the GoalID column name. This means the Rating table uses a composite primary key that combines the EvaluationID column and the GoalID column.

Accounting

The GDS accounting department is responsible for seeing that the company is paid for each package it delivers. GDS invoices its customers for each delivery completed. The invoices are sent to the customer and payment is requested within 30 days.

Even though GDS delivers its customers' packages at the speed of light, those same customers pay GDS at a much slower speed. "Molasses at the northern pole of Antares Prime" was the analogy used by the current Chief Financial Droid. Therefore, GDS must track when invoices are paid, how much was paid, and how much is still outstanding.

Figure 3-22 shows the tables in the Galactic database used by the accounting department. Notice the Customer table appears in both Figure 3-20 and Figure 3-22.

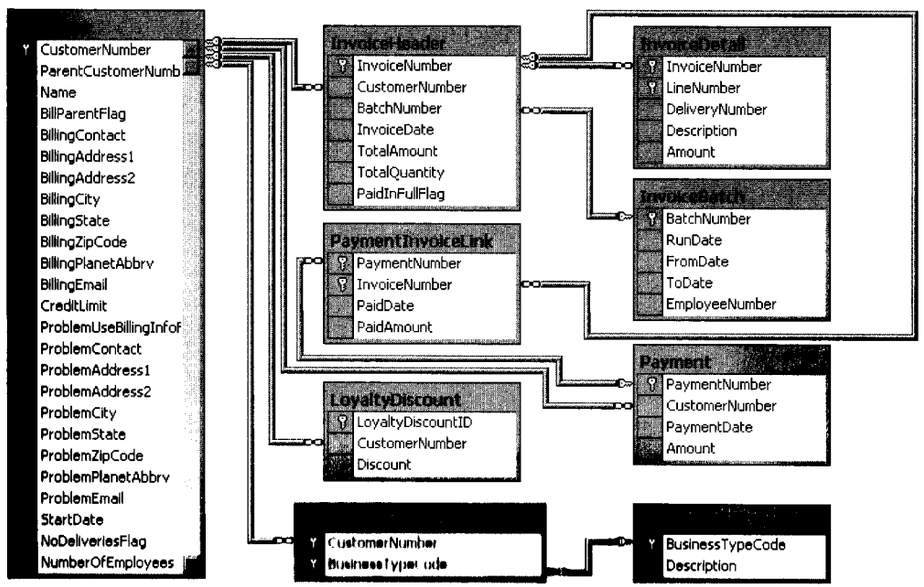


Figure 3-22 *The accounting department tables from the Galactic database*

This is the same table in both diagrams. This table is shown in both, because it is a major part of both the package tracking and the accounting business processes.

Transport Maintenance

In addition to all this, GDS must maintain a fleet of transports. Careful records are kept on the repair and preventative maintenance work done on each transport. GDS also has a record of each flight a transport makes, as well as any accidents and mishaps involved.

Maintenance records are extremely important, not only to GDS itself, but also to the Federation Space Flight Administration (FSFA). Without proper maintenance records on all its transports, GDS would be shut down by the FSFA in a nanosecond. You may think this is an exaggeration, but the bureaucratic androids at the FSFA have extremely high clock rates.

Figure 3-23 shows the transport maintenance tables in the Galactic database.

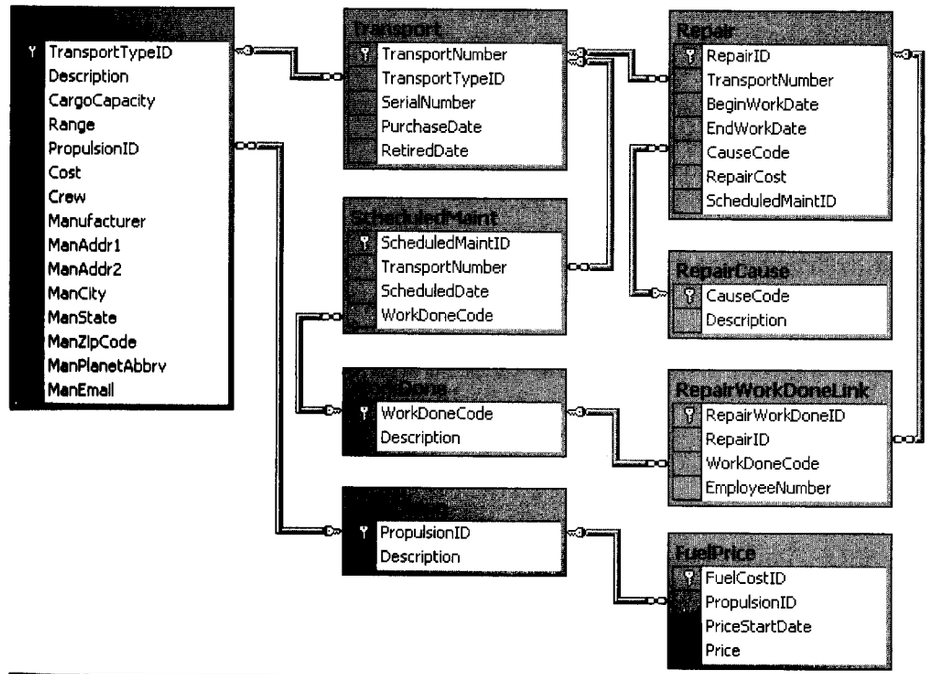


Figure 3-23 *The transport maintenance tables from the Galactic database*

Querying Data

You have now looked at the database concepts of normalization, relations, and joins. You have also been introduced to the Galactic database. We use this relational database throughout the remainder of this book for our examples. Now, it is time to look more specifically at how you retrieve the data from the database into a format you can use for reporting. This is done through the database query.

A *query* is a request for some action on the data in one or more tables. An *INSERT query* adds one or more rows to a database table. An *UPDATE query* modifies the data in one or more existing rows of a table. A *DELETE query* removes one or more rows from a table. Because we are primarily interested in retrieving data for reporting, the query we are going to concern ourselves with is the *SELECT query*, which reads data from one or more tables (it does not add, update, or delete data).

We will look at the various parts of the SELECT query. This is to help you become familiar with this important aspect of reporting. The good news is Reporting Services provides a tool to guide you through the creation of queries, including the SELECT query. That tool is the Query Designer.

If you are familiar with SELECT queries and are more comfortable typing your queries from scratch, you can bypass the Query Designer and type in your queries directly. If SELECT queries are new to you, the following section can help you become familiar with the SELECT query and what it can do for you. Rest assured, the Query Designer enables you to take advantage of all the features of the SELECT query without having to memorize syntax or type a lot of code.

NOTE

If you have another query-creation tool you like to use instead of the Query Designer, you can create your queries with that tool, and then copy them into the appropriate locations in the report definition.

The SELECT Query

The SELECT query is used to retrieve data from tables in the database. When a SELECT query is run, it returns a result set containing the selected data. With few exceptions, your reports will be built on result sets created by SELECT queries.

The SELECT query is often referred to as a *SELECT statement*. One reason for this is because it can be read like an English sentence or statement. As with a sentence in English, a SELECT statement is made up of clauses that modify the meaning of the statement.

The various parts, or clauses, of the SELECT statement enable you to control the data contained in the result set. Use the *FROM clause* to specify which table the data will be selected from. The *FIELD LIST* permits you to choose the columns that will appear in the result set. The *JOIN clause* lets you specify additional tables that will be joined with the table in the FROM clause to contribute data to the result set. The *WHERE clause* enables you to set conditions that determine which rows will be included in the result set. Finally, you can use the *ORDER BY clause* to sort the result set, and the *GROUP BY clause* and the *HAVING clause* to combine detail rows into summary rows.

NOTE

The query statements shown in the remainder of this chapter all use the Galactic database. If you want to try out the various query statements as they are being discussed, open a query window for the Galactic database in the SQL Server Management Studio. If you are not familiar with SQL Server Management Studio, you can try out the queries in the Reporting Services Generic Query Designer. To do this, turn to Chapter 5 and follow the steps for Task 1 of the Transport List Report, but stop after Step 26. You will be in the Generic Query Designer. You can enter the query statements in the upper portion of the Generic Query Designer and execute them by clicking the toolbar button with the exclamation point. When you are finished, close the application without saving your changes.

The FROM Clause

The SELECT statement in its simplest form includes only a FROM clause. Here is a SELECT statement that retrieves all rows and all columns from the Customer table:

```
SELECT *
FROM dbo.Customer
```

The word "SELECT" is required to let the database know this is going to be a SELECT query, as opposed to an INSERT, UPDATE, or DELETE query. The asterisk (*) means all columns will be included in the result set. The remainder of the statement is the FROM clause. It says the data is to be selected from the Customer table. We discuss the meaning of dbo. in a moment.

As stated earlier, the SELECT statement can be read as if it were a sentence. This SELECT statement is read, "Select all columns from the Customer table." If we run this SELECT statement in the Galactic database, the results would appear similar to Figure 3-24. The SELECT query is being run in the Query Designer window of Visual Studio. Note, the scroll bars on the right and on the bottom of the result set area indicate not all the rows and columns returned can fit on the screen.

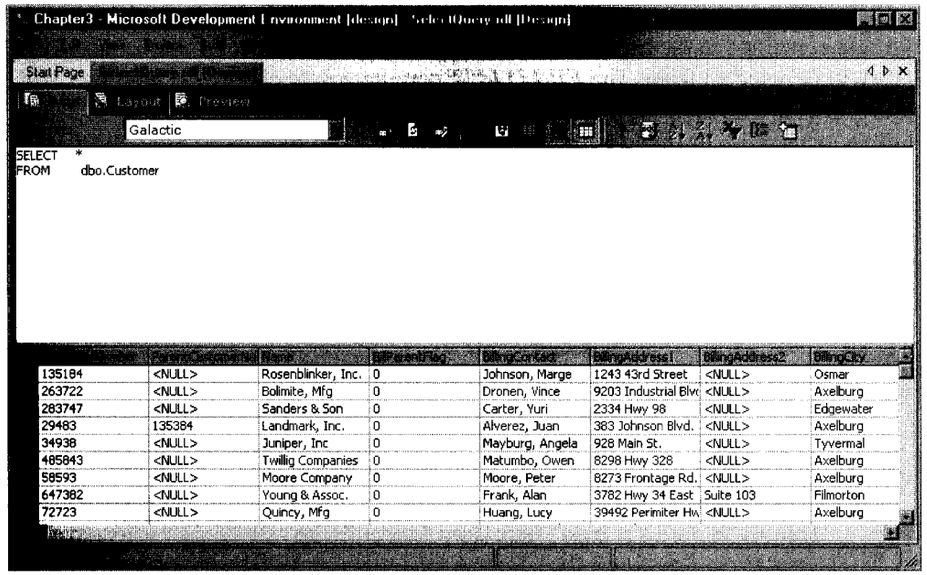


Figure 3-24 *The SELECT statement in its simplest form*

Note, the table name, Customer, has dbo. in front of it. The dbo is the name of the owner of the table. Usually this is the user who created the table. Here, dbo stands for database owner, meaning the user who owns the database is also the user who owns the table. The dbo abbreviation is also another name for the system administrator login. In many cases, an administrative user, logged into the database, will create the database tables. Because of this, the table owner will more than likely be dbo.

In the Galactic database, the dbo.Customer table was created by the system administrator. If another user with a database login of User2 also has rights to create tables in the Galactic database, they could also create a Customer table. This second table would be known as User2.Customer.

This situation, with two tables of the same name in the same database, does not happen often and is probably not a great idea. It can quickly lead to confusion and errors. Even though this is a rare occurrence, the Query Designer needs to account for this situation. The Query Designer uses both the name of the table owner and the name of the table itself in the queries it builds and executes for you.

The FIELD LIST

In the previous example, the SELECT statement contained all the columns in the table. When creating reports, you only

need to work with some of the columns of a table in any given result set. Including all the columns in a result set when only a few columns are required wastes computing power and network bandwidth.

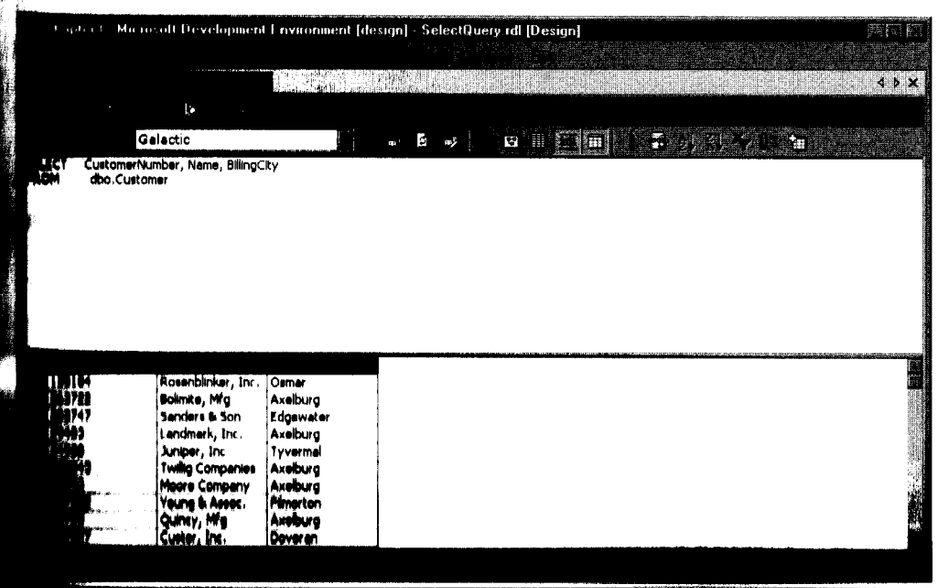
A FIELD LIST provides the capability you need to specify which columns to include in the result set. When a FIELD LIST is added to the SELECT statement, it appears similar to the following:

```
SELECT CustomerNumber, Name, BillingCity
FROM dbo.Customer
```

The bold portion of the SELECT statement indicates changes from the previous SELECT statement.

This statement returns only the CustomerNumber, Name, and BillingCity columns from the Customer table. The result set created by this SELECT statement is shown in Figure 3-25.

In addition to the names of the fields to include in the result set, the FIELD LIST can contain a word that influences the number of rows in the result set. Usually, there is one row in the result set for each row in the table from which you are selecting data. However, this can be changed by adding the word "DISTINCT" at the beginning of the FIELD LIST.



A SELECT statement with a FIELD LIST

When you use DISTINCT in the FIELD LIST, you are saying that you only want one row in the result set for each distinct set of values. In other words, the result set from a DISTINCT query will not have any two rows that have exactly the same values in every column. Here is an example of a DISTINCT query:

```
SELECT DISTINCT BillingCity
FROM dbo.Customer
```

This query returns a list of all the billing cities in the Customer table. A number of customers have the same billing city, but these duplicates have been removed from the result set, as shown in Figure 3-26.

The JOIN Clause

When your database is properly normalized, you are likely to need data from more than one table to fulfill your reporting requirements. As discussed earlier in this chapter, the way to get information from more than one table is to use a join. The JOIN clause in the SELECT statement enables you to include a join of two or more tables in your result set.

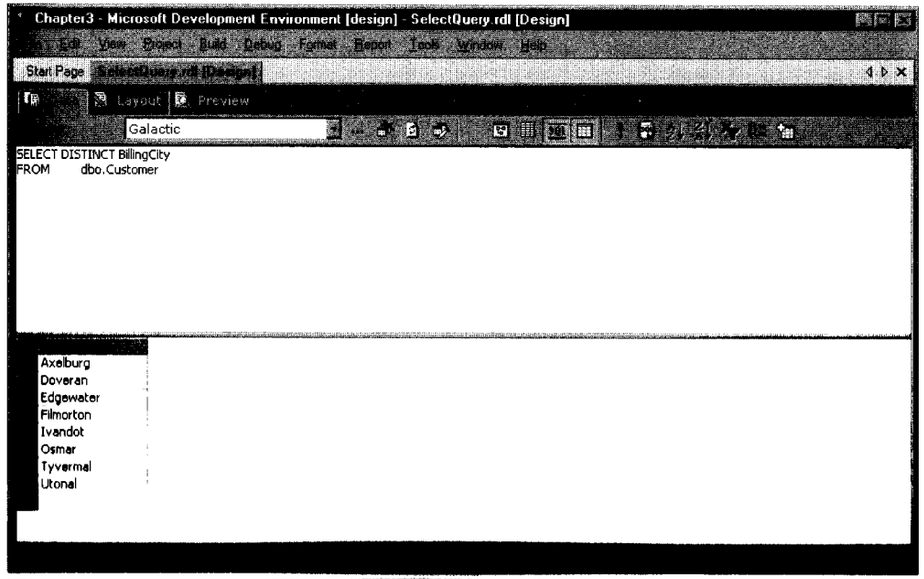


Figure 3-26 *A DISTINCT query*

The first part of the JOIN clause specifies which table is being joined. The second part determines the two columns that are linked to create the join. Joining the Invoice Header table to the Customer table looks like this:

```
SELECT dbo.Customer.CustomerNumber,
    dbo.Customer.Name,
    dbo.Customer.BillingCity,
    dbo.InvoiceHeader.InvoiceNumber,
    dbo.InvoiceHeader.TotalAmount
FROM dbo.Customer
INNER JOIN dbo.InvoiceHeader
 ON dbo.Customer.CustomerNumber = dbo.InvoiceHeader.CustomerNumber
```

With the Customer table and the Invoice Header table joined, you have a situation where some columns in the result set have the same name. For example, a CustomerNumber column is in the Customer table and a CustomerNumber column is in the Invoice Header table. When you use the FIELD LIST to tell the database which fields to include in the result set, you need to uniquely identify these fields using both the table name and the column name.

If you do not do this, the query will not run and you will receive an error. Nothing prevents you from using the table name in front of each column name, whether it is a duplicate or not, as in this example. Using the table name in front of each column name makes it immediately obvious where every column in the result set is selected from. The result set created by this SELECT statement is shown in Figure 3-27.

You can add a third table to the query by adding another JOIN clause to the SELECT statement. This additional table can be joined to the table in the FROM clause or to the table in the first JOIN clause. In this statement, we add the Loyalty Discount table and join it to the Customer table:

```
SELECT dbo.Customer.CustomerNumber,
    dbo.Customer.Name,
    dbo.Customer.BillingCity,
    dbo.InvoiceHeader.InvoiceNumber,
    dbo.InvoiceHeader.TotalAmount,
    dbo.LoyaltyDiscount.Discount
FROM dbo.Customer
INNER JOIN dbo.InvoiceHeader
 ON dbo.Customer.CustomerNumber = dbo.InvoiceHeader.CustomerNumber
INNER JOIN dbo.LoyaltyDiscount
 ON dbo.Customer.CustomerNumber = dbo.LoyaltyDiscount.CustomerNumber
```

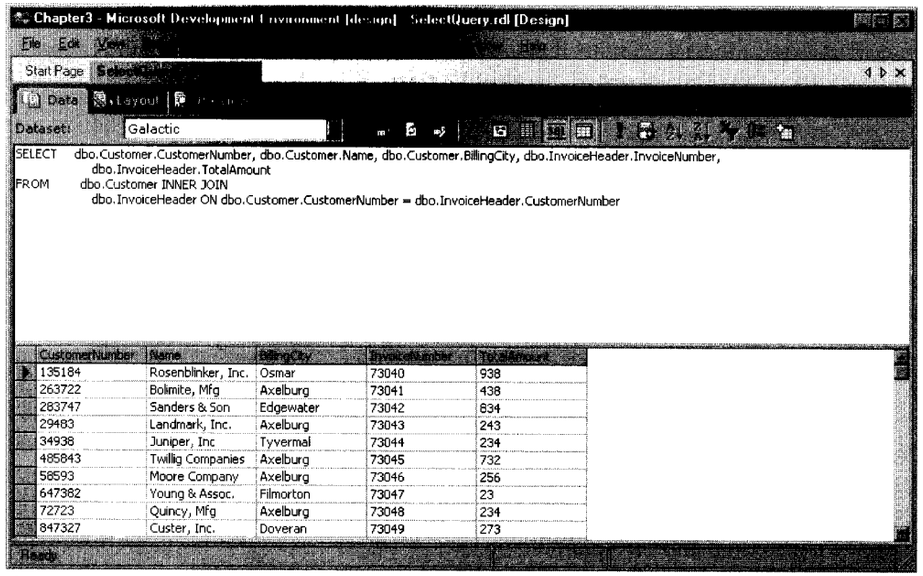


Figure 3-27 *A SELECT statement with a JOIN clause*

The result set from this SELECT statement is shown in Figure 3-28. Notice that the result set is rather small. This is because Landmark, Inc. is the only customer currently receiving a loyalty discount. Because an INNER JOIN was used to add the Loyalty Discount table, only customers that have a loyalty discount are included in the result set.

To make our result set a little more interesting, let's try joining the Loyalty Discount table with an OUTER JOIN rather than an INNER JOIN. Here is the same statement, except the Customer table is joined to the Loyalty Discount table with a LEFT OUTER JOIN:

```
SELECT dbo.Customer.CustomerNumber,
    dbo.Customer.Name,
    dbo.Customer.BillingCity,
    dbo.InvoiceHeader.InvoiceNumber,
    dbo.InvoiceHeader.TotalAmount,
    dbo.LoyaltyDiscount.Discount
FROM dbo.Customer
INNER JOIN dbo.InvoiceHeader
 ON dbo.Customer.CustomerNumber = dbo.InvoiceHeader.CustomerNumber
LEFT OUTER JOIN dbo.Loy
 ON dbo.Customer.Cust                    tyDiscount.CustomerNumber
```

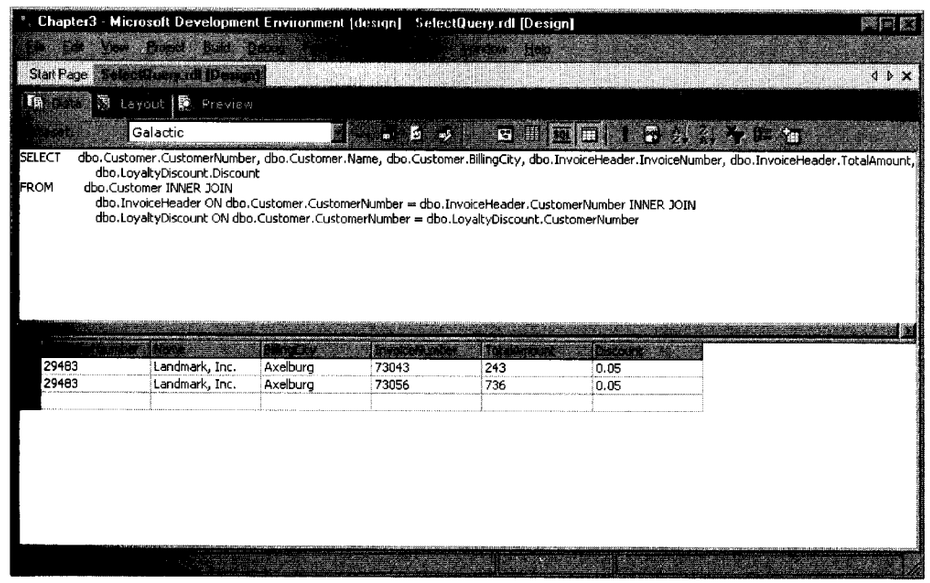



Figure 3-28 *A SELECT statement with two JOIN clauses*

The result set for this SELECT statement is shown in Figure 3-29. Notice that the value for the Discount column is NULL in the rows for all the customers except for Landmark, Inc. This is to be expected because there is no record in the Loyalty Discount table to join with these customers. When no value is in a column, the result set will contain a NULL value.

The WHERE Clause

Up to this point, the result sets have included all the rows in the table or all the rows that result from the joins. The FIELD LIST limits which columns are being returned in the result set. Nothing, however, placed a limit on the rows.

To limit the number of rows in the result set, you need to add a WHERE clause to your SELECT statement. The WHERE clause includes one or more logical expressions that must be true for a row before it can be included in the result set. Here is an example of a SELECT statement with a WHERE clause:

```
SELECT dbo.Customer.CustomerNumber,
    dbo.Customer.Name,
    dbo.Customer.BillingCity,
    dbo.InvoiceHeader.InvoiceNumber,
    dbo.InvoiceHeader.TotalAmount,
    dbo.LoyaltyDiscount.Discount
```

```
FROM dbo.Customer
INNER JOIN dbo.InvoiceHeader
 ON dbo.Customer.CustomerNumber = dbo.InvoiceHeader.CustomerNumber
LEFT OUTER JOIN dbo.LoyaltyDiscount
 ON dbo.Customer.CustomerNumber = dbo.LoyaltyDiscount.CustomerNumber
WHERE (dbo.Customer.BillingCity = 'Axelburg')
```

The word 'Axelburg' (enclosed in single quotes) is a string constant. A *string constant*, also known as a *string literal*, is an actual text value. The string constant instructs SQL Server to use the text between the single quotes as a value rather than the name of a column or a table. In this example, only customers with a value of Axelburg in their BillingCity column will be included in the result set, as shown in Figure 3-30.

NOTE

Microsoft SQL Server 2005, in its standard configuration, insists on single quotes around string constants, such as 'Axelburg' in the previous SELECT statement. SQL Server 2005 assumes that anything enclosed in double quotes is a field name.

To create more complex criteria for your result set, you can have multiple logical expressions in the WHERE clause. The logical expressions are linked together with

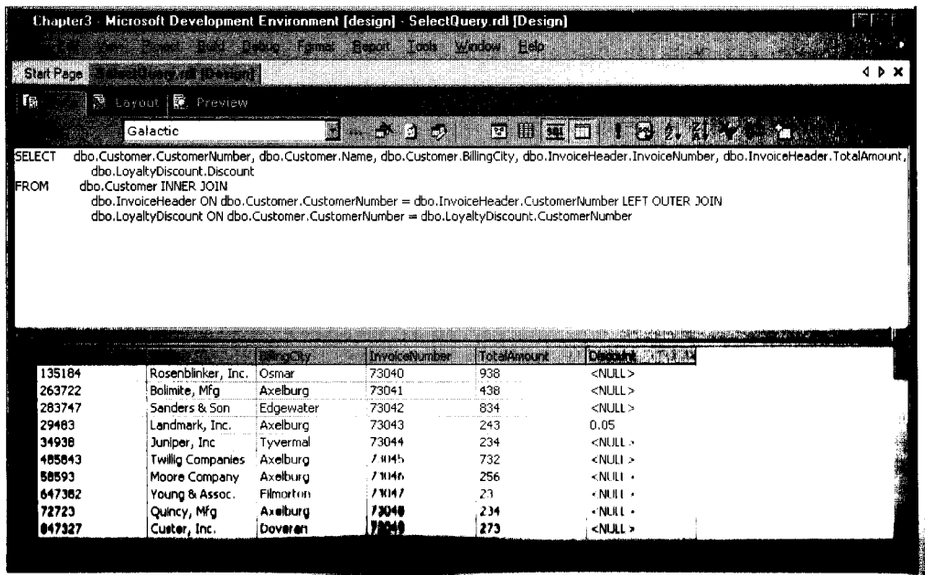


Figure 3-29 *A SELECT statement with a JOIN and an OUTER JOIN*

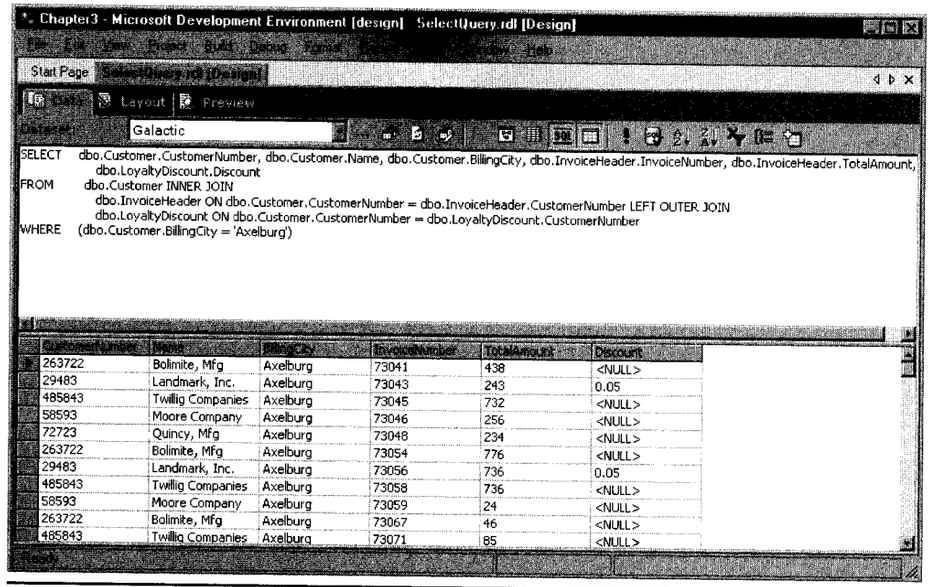


Figure 3-30 *A SELECT statement with a WHERE clause*

an AND or an OR. When an AND is used to link logical expressions, the logical expressions on both sides of the AND must be true for a row in order for that row to be included in the result set. When an OR is used to link two logical expressions, either one or both of the logical expressions must be true for a row in order for that row to be included in the result set.

This SELECT statement has two logical expressions:

```
SELECT dbo.Customer.CustomerNumber,
    dbo.Customer.Name,
    dbo.Customer.BillingCity,
    dbo.InvoiceHeader.InvoiceNumber,
    dbo.InvoiceHeader.TotalAmount,
    dbo.LoyaltyDiscount.Discount
FROM dbo.Customer
INNER JOIN dbo.InvoiceHeader
 ON dbo.Customer.CustomerNumber = dbo.InvoiceHeader.CustomerNumber
LEFT OUTER JOIN dbo.LoyaltyDiscount
 ON dbo.Customer.CustomerNumber = dbo.LoyaltyDiscount.CustomerNumber
WHERE (dbo.Customer.BillingCity = 'Axelburg')
AND (dbo.Customer.Name > 'G')
```

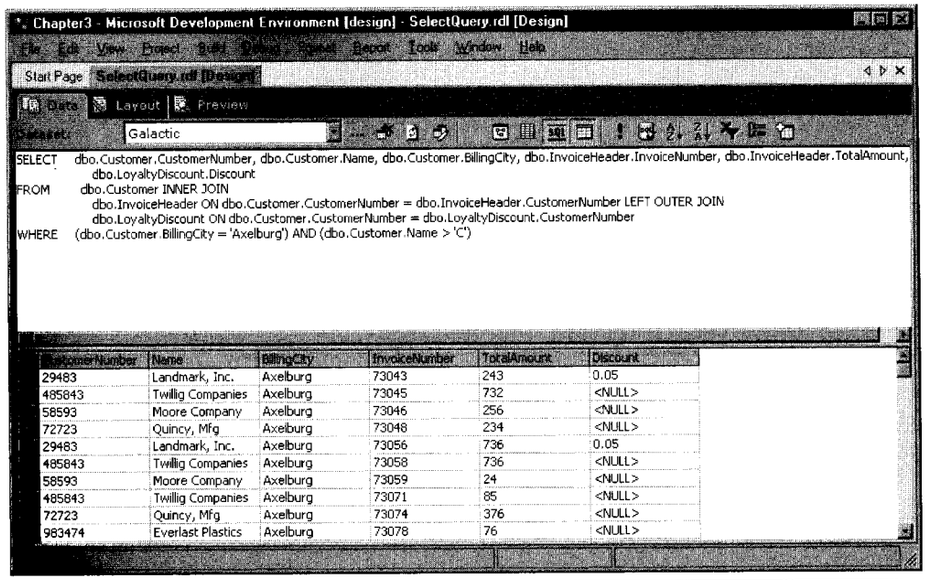


Figure 3-31 *A SELECT statement with two logical expressions in the WHERE clause*

Only customers with a value of Axelburg in their BillingCity column and with a name that comes after *C* will be included in the result set. This result set is shown in Figure 3-31.

The ORDER BY Clause

Up to this point, the data in the result sets has shown up in any order it pleases. As discussed previously, this will probably not be acceptable for most reports. You can add an ORDER BY clause to your SELECT statement to obtain a sorted result set. This statement includes an ORDER BY clause with multiple columns:

```
SELECT dbo.Customer.CustomerNumber,
    dbo.Customer.Name,
    dbo.Customer.BillingCity,
    dbo.InvoiceHeader.InvoiceNumber,
    dbo.InvoiceHeader.TotalAmount,
    dbo.LoyaltyDiscount.Discount
FROM dbo.Customer
INNER JOIN dbo.InvoiceHeader
```



```
 ON dbo.Customer.CustomerNumber = dbo.InvoiceHeader.CustomerNumber
LEFT OUTER JOIN dbo.LoyaltyDiscount
 ON dbo.Customer.CustomerNumber = dbo.LoyaltyDiscount.CustomerNumber
WHERE (dbo.Customer.BillingCity = 'Axelburg')
AND (dbo.Customer.Name > 'C')
ORDER BY dbo.Customer.Name DESC, dbo.InvoiceHeader.InvoiceNumber
```

The result set created by this SELECT statement, shown in Figure 3-32, is first sorted by the contents of the Name column in the Customer table. The DESC that follows dbo.Customer.Name in the ORDER BY clause specifies the sort order for the customer name sort. DESC means this sort is done in descending order. In other words, the customer names will be sorted from the end of the alphabet to the beginning.

Several rows have the same customer name. For this reason, a second sort column is specified. This second sort is only applied within each group of identical customer names. For example, Twillig Companies has three rows in the result set. These three rows are sorted by the second sort, which is invoice number. No sort order is specified for the invoice number sort, so this defaults to an ascending sort. In other words, the invoice numbers are sorted from lowest to highest.

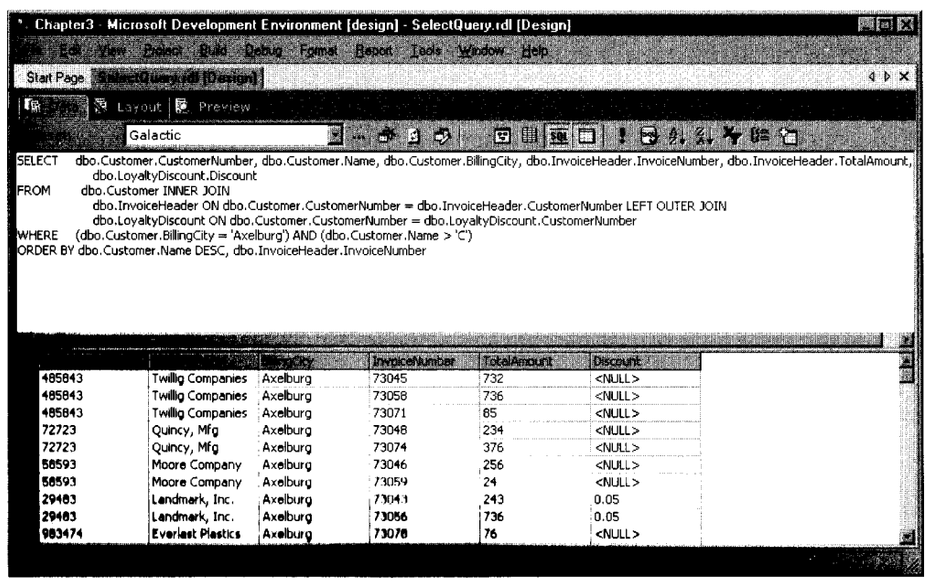


Figure 3-32 *A SELECT statement with an ORDER BY clause*

Constant and Calculated Fields

Our SELECT statement examples thus far have used an asterisk symbol or a FIELD LIST that includes only columns. A FIELD LIST can, in fact, include other things as well. For example, a FIELD LIST can include a constant value, as is shown here:

```
SELECT dbo.Customer.CustomerNumber,
    dbo.Customer.Name,
    dbo.Customer.BillingCity,
    dbo.InvoiceHeader.InvoiceNumber,
    dbo.InvoiceHeader.TotalAmount,
    dbo.LoyaltyDiscount.Discount,
    'AXEL' AS ProcessingCode
FROM dbo.Customer
INNER JOIN dbo.InvoiceHeader
 ON dbo.Customer.CustomerNumber = dbo.InvoiceHeader.CustomerNumber
LEFT OUTER JOIN dbo.LoyaltyDiscount
 ON dbo.Customer.CustomerNumber = dbo.LoyaltyDiscount.CustomerNumber
WHERE (dbo.Customer.BillingCity = 'Axelburg')
AND (dbo.Customer.Name > 'C')
ORDER BY dbo.Customer.Name DESC, dbo.InvoiceHeader.InvoiceNumber
```

The string constant 'AXEL' has been added to the FIELD LIST. This creates a new column in the result set with the value AXEL in each row. By including AS ProcessingCode on this line, we give this result set column a column name of ProcessingCode. Constant values of other data types, such as dates or numbers, can also be added to the FIELD LIST. The result set for this SELECT statement is shown in Figure 3-33.

In addition to adding constant values, you can also include calculations in the FIELD LIST. This SELECT statement calculates the discounted invoice amount based on the total amount of the invoice and the loyalty discount:

```
SELECT dbo.Customer.CustomerNumber,
    dbo.Customer.Name,
    dbo.Customer.BillingCity,
    dbo.InvoiceHeader.InvoiceNumber,
    dbo.InvoiceHeader.TotalAmount,
    dbo.LoyaltyDiscount.Discount,
    dbo.InvoiceHeader.TotalAmount -
      (dbo.InvoiceHeader.TotalAmount *
            dbo.LoyaltyDiscount.Discount)
                AS Discounted
```



```
FROM dbo.Customer
INNER JOIN dbo.InvoiceHeader
 ON dbo.Customer.CustomerNumber = dbo.InvoiceHeader.CustomerNumber
LEFT OUTER JOIN dbo.LoyaltyDiscount
 ON dbo.Customer.CustomerNumber = dbo.LoyaltyDiscount.CustomerNumber
WHERE (dbo.Customer.BillingCity = 'Axelburg')
AND (dbo.Customer.Name > 'C')
ORDER BY dbo.Customer.Name DESC, dbo.InvoiceHeader.InvoiceNumber
```

The result set for this SELECT statement is shown in Figure 3-34. Notice the value for the calculated column, DiscountedTotalAmount, is NULL for all the rows that are not for Landmark, Inc. This is because we are using the value of the Discount column in our calculation. The Discount column has a value of NULL for every row except for the Landmark, Inc. rows.

A NULL value cannot be used successfully in any calculation. Any time you try to add, subtract, multiply, or divide a number by NULL, the result is NULL. The only way to receive a value in these situations is to give the database a valid value to use in

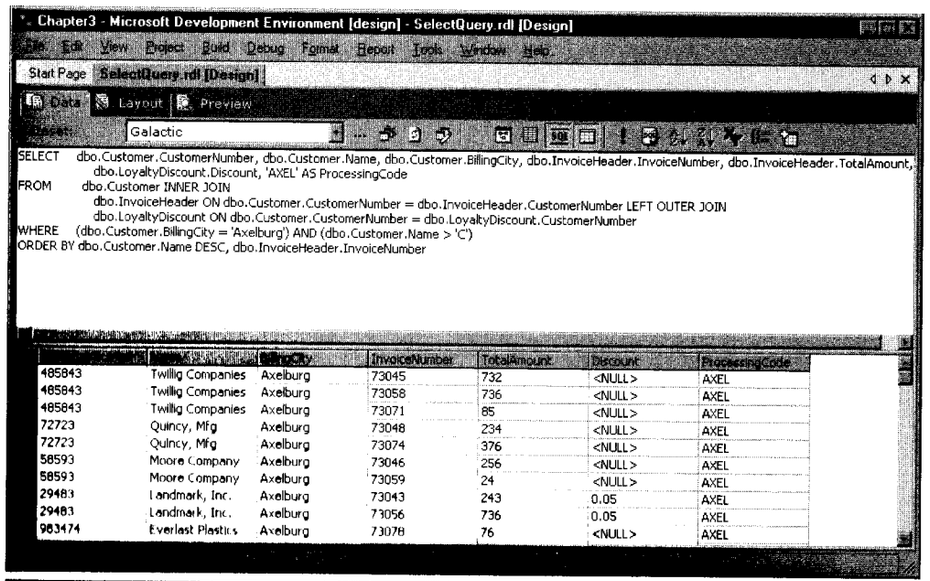
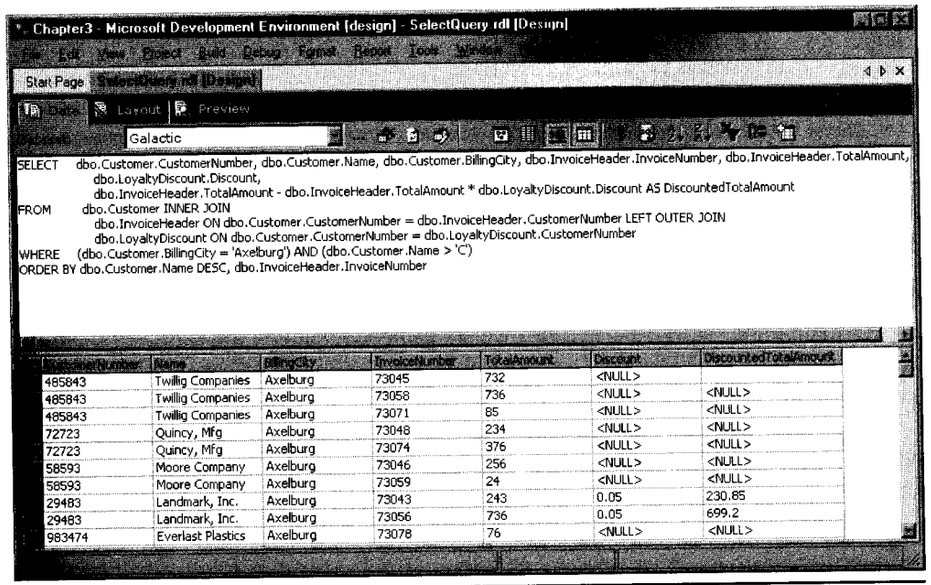


Figure 3-33 *A SELECT statement with a constant in the FIELD LIST*



CustomerNumber	Name	BillingCity	InvoiceNumber	TotalAmount	Discount	DiscountedTotalAmount
485843	Twillig Companies	Axelburg	73045	732	<NULL>	
485843	Twillig Companies	Axelburg	73058	736	<NULL>	<NULL>
485843	Twillig Companies	Axelburg	73071	85	<NULL>	<NULL>
72723	Quincy, Mfg	Axelburg	73048	234	<NULL>	<NULL>
72723	Quincy, Mfg	Axelburg	73074	376	<NULL>	<NULL>
58593	Moore Company	Axelburg	73046	256	<NULL>	<NULL>
58593	Moore Company	Axelburg	73059	24	<NULL>	<NULL>
29483	Landmark, Inc.	Axelburg	73043	243	0.05	230.85
29483	Landmark, Inc.	Axelburg	73056	736	0.05	699.2
983474	Everlast Plastics	Axelburg	73078	76	<NULL>	<NULL>

Figure 3-34 *A SELECT statement with a calculated column in the FIELD LIST*

place of any NULLs it might encounter. This is done using the ISNULL( ) function, as shown in the following statement:

```
SELECT dbo.Customer.CustomerNumber,
    dbo.Customer.Name,
    dbo.Customer.BillingCity,
    dbo.InvoiceHeader.InvoiceNumber,
    dbo.InvoiceHeader.TotalAmount,
    dbo.LoyaltyDiscount.Discount,
    dbo.InvoiceHeader.TotalAmount -
      (dbo.InvoiceHeader.TotalAmount *
        ISNULL(dbo.LoyaltyDiscount.Discount,0.00))
                    AS DiscountedTotalAmount
FROM dbo.Customer
INNER JOIN dbo.InvoiceHeader
 ON dbo.Customer.CustomerNumber = dbo.InvoiceHeader.CustomerNumber
```

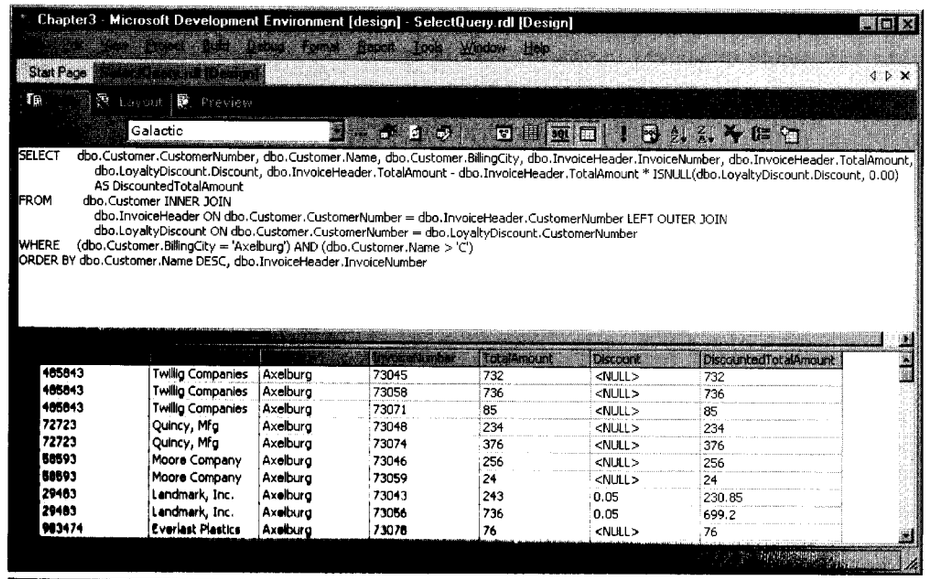


```
LEFT OUTER JOIN dbo.LoyaltyDiscount
 ON dbo.Customer.CustomerNumber = dbo.LoyaltyDiscount.CustomerNumber
WHERE (dbo.Customer.BillingCity = 'Axelburg')
AND (dbo.Customer.Name > 'C')
ORDER BY dbo.Customer.Name DESC, dbo.InvoiceHeader.InvoiceNumber
```

Now, when the database encounters a NULL value in the Discount column while it is performing the calculation, it substitutes a value of 0.00 and continues with the calculation. The database only performs this substitution when it encounters a NULL value. If any other value is in the Discount column, it uses that value. The result set from this SELECT statement is shown in Figure 3-35.

The GROUP BY Clause

Our sample SELECT statement appears to resemble a run-on sentence. You have seen, however, that each of these clauses is necessary to change the meaning of the statement and to provide the desired result set. We will add just two more clauses to the sample SELECT statement before we are done.



485843	Twillig Companies	Axelburg	73045	732	<NULL>	732
485843	Twillig Companies	Axelburg	73058	736	<NULL>	736
485843	Twillig Companies	Axelburg	73071	85	<NULL>	85
72723	Quincy, Mfg	Axelburg	73048	234	<NULL>	234
72723	Quincy, Mfg	Axelburg	73074	376	<NULL>	376
58593	Moore Company	Axelburg	73046	256	<NULL>	256
58593	Moore Company	Axelburg	73059	24	<NULL>	24
29483	Landmark, Inc.	Axelburg	73043	243	0.05	230.85
29483	Landmark, Inc.	Axelburg	73056	736	0.05	699.2
983474	Everlast Plastics	Axelburg	73078	76	<NULL>	76

Figure 3-35 *A SELECT statement using the ISNULL() function*

At times, as you are analyzing data, you only want to see information at a summary level, rather than viewing all the detail. In other words, you want the result set to group together the information from several rows to form a summary row. Additional instructions must be added to our SELECT statement in two places for this to happen.

First, you need to specify which columns are going to be used to determine when a summary row will be created. These columns are placed in the GROUP BY clause. Consider the following SELECT statement:

```
SELECT dbo.Customer.CustomerNumber,
    dbo.Customer.Name,
    dbo.Customer.BillingCity,
    COUNT(dbo.InvoiceHeader.InvoiceNumber) AS NumberOfInvoices,
    SUM(dbo.InvoiceHeader.TotalAmount) AS TotalAmount,
    dbo.LoyaltyDiscount.Discount,
    SUM(dbo.InvoiceHeader.TotalAmount -
      (dbo.InvoiceHeader.TotalAmount *
        ISNULL(dbo.LoyaltyDiscount.Discount,0.00)))
                    AS DiscountedTotalAmount
FROM dbo.Customer
INNER JOIN dbo.InvoiceHeader
 ON dbo.Customer.CustomerNumber = dbo.InvoiceHeader.CustomerNumber
LEFT OUTER JOIN dbo.LoyaltyDiscount
 ON dbo.Customer.CustomerNumber = dbo.LoyaltyDiscount.CustomerNumber
WHERE (dbo.Customer.BillingCity = 'Axelburg')
AND (dbo.Customer.Name > 'C')
GROUP BY dbo.Customer.CustomerNumber, dbo.Customer.Name,
     dbo.Customer.BillingCity, dbo.LoyaltyDiscount.Discount
ORDER BY dbo.Customer.Name DESC
```

The CustomerNumber, Name, BillingCity, and Discount columns are included in the GROUP BY clause. When this query is run, each unique set of values from these four columns will result in a row in the result set.

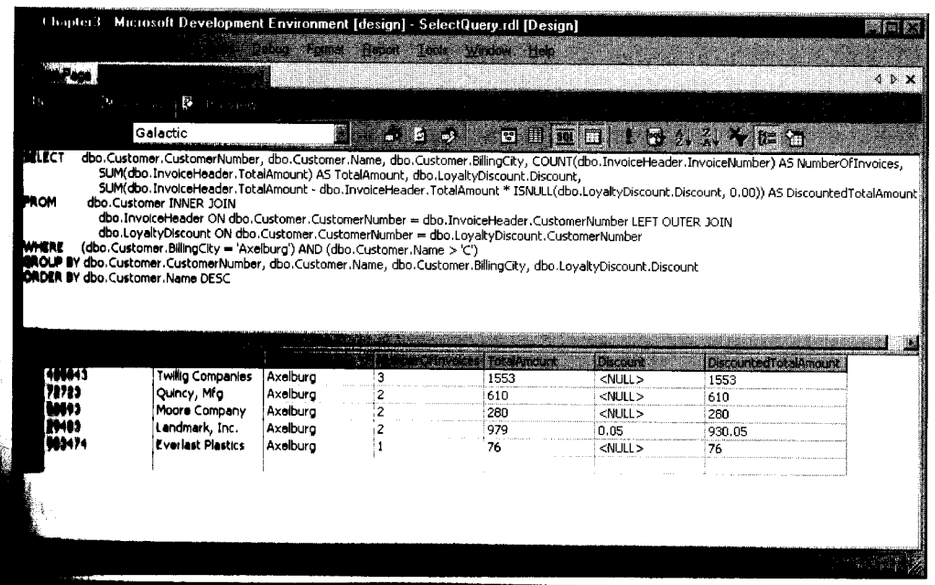
Second, you need to specify how the columns in the FIELD LIST that are not included in the GROUP BY clause are to be handled. In the sample SELECT statement, the InvoiceNumber and TotalAmount columns are in the FIELD LIST, but are not part of the GROUP BY clause. The calculated column, DiscountedTotalAmount, is also in the FIELD LIST, but it is not present in the GROUP BY clause. In the sample SELECT statement, these three columns are the non-group-by columns.

The SELECT statement is asking for the values from several rows to be combined into one summary row. The SELECT statement needs to provide a way for this combining to take place. This is done by enclosing each non-group-by column in a special function called an aggregation, which performs a mathematical

operation on values from a number of rows and returns a single result. Aggregate functions include:

- **SUM( )** Returns the sum of the values
- **AVG( )** Returns the average of the values
- **COUNT( )** Returns a count of the values
- **MAX( )** Returns the largest value
- **MIN( )** Returns the smallest value

The SELECT statement in our group by example uses the SUM( ) aggregate function to return the sum of the invoice amount and the sum of the discounted amount for each customer. It also uses the COUNT( ) aggregate function to return the number of invoices for each customer. The result set from this SELECT statement is shown in Figure 3-36. Note, when an aggregate function is placed around a column name in the FIELD LIST, the SELECT statement can no longer determine what name to use for that column in the result set. You need to supply a column name to use in the result set, as shown in this SELECT statement.



3-36 A SELECT statement with a GROUP BY clause

NOTE

When you're using a GROUP BY clause, all columns in the FIELD LIST must either be included in the GROUP BY clause or be enclosed in an aggregate function. In the sample SELECT statement, the CustomerNumber column is all that is necessary in the GROUP BY clause to provide the desired grouping. However, because the Name, BillingCity, and Discount columns do not lend themselves to being aggregated, they are included in the GROUP BY clause along with the CustomerNumber column.

The HAVING Clause

The GROUP BY clause has a special clause that can be used with it to determine which grouped rows will be included in the result set. This is the HAVING clause. The HAVING clause functions similarly to the WHERE clause. The *WHERE* clause limits the rows in the result set by checking conditions at the row level. The *HAVING* clause limits the rows in the result set by checking conditions at the group level.

Consider the following SELECT statement:

```
SELECT dbo.Customer.CustomerNumber,
    dbo.Customer.Name,
    dbo.Customer.BillingCity,
    COUNT(dbo.InvoiceHeader.InvoiceNumber) AS NumberOfInvoices,
    SUM(dbo.InvoiceHeader.TotalAmount) AS TotalAmount,
    dbo.LoyaltyDiscount.Discount,
    SUM(dbo.InvoiceHeader.TotalAmount -
      (dbo.InvoiceHeader.TotalAmount *
        ISNULL(dbo.LoyaltyDiscount.Discount,0.00)))
                    AS DiscountedTotalAmount
FROM dbo.Customer
INNER JOIN dbo.InvoiceHeader
 ON dbo.Customer.CustomerNumber = dbo.InvoiceHeader.CustomerNumber
LEFT OUTER JOIN dbo.LoyaltyDiscount
 ON dbo.Customer.CustomerNumber = dbo.LoyaltyDiscount.CustomerNumber
WHERE (dbo.Customer.BillingCity = 'Axelburg')
AND (dbo.Customer.Name > 'C')
GROUP BY dbo.Customer.CustomerNumber, dbo.Customer.Name,
     dbo.Customer.BillingCity, dbo.LoyaltyDiscount.Discount
HAVING COUNT(dbo.InvoiceHeader.InvoiceNumber) >= 2
ORDER BY dbo.Customer.Name DESC
```

The WHERE clause says that a row must have a BillingCity column with a value of Axelburg and a Name column with a value greater than *C* before it can be included in the group. The HAVING clause says a group must contain at least two invoices before it can be included in the result set for this SELECT statement is shown in Figure 3-37.

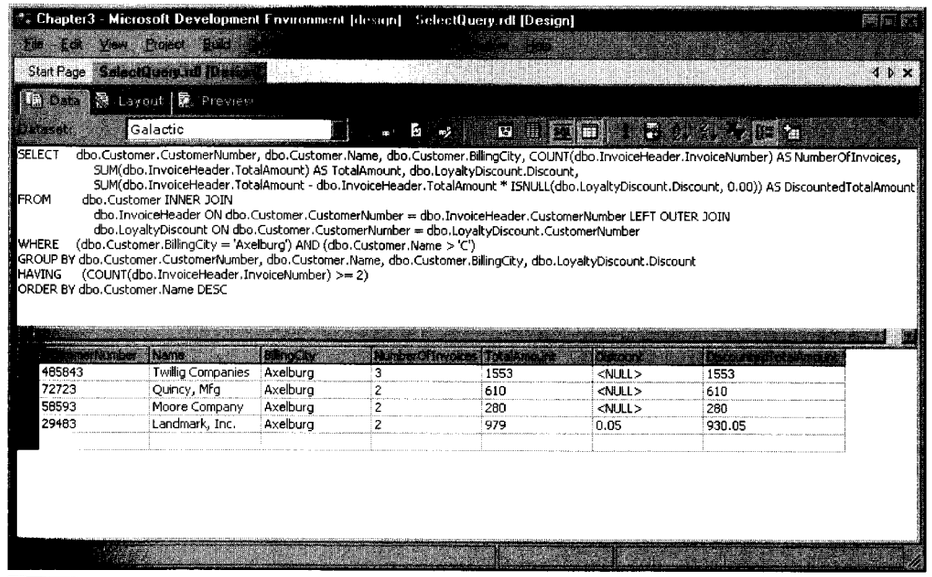


Figure 3-37 *A SELECT statement with a HAVING clause*

On to the Reports

Good reporting depends more on getting the right data out of the database than it does on creating a clean report design and delivering the report in a timely manner. If you are feeling a little overwhelmed by the workings of relational databases and SELECT queries, don't worry. Refer to this chapter from time to time if you need to.

Also, remember, Reporting Services provides you with the Query Designer tool to assist with the query-creation process. You needn't remember the exact syntax for the LEFT OUTER JOIN or a GROUP BY clause. What you do need to know are the capabilities of the SELECT statement, so you know what to instruct the Query Designer to create.

Finally, when you are creating your queries, use the same method that was used here: In other words, build them one step at a time. Join together the tables you will need for your report, determine what columns are required, and then come up with the WHERE clause that gets you only the rows you are looking for. After that, you can add in the sorting and grouping. Assemble one clause, and then another and another, and pretty soon, you will have a slam-bang query that will give you exactly the data you need!

Now, on to the reports. . . .