Chapter 17 Advantage and Delphi

Note: This chapter accompanies the book Advantage Database Server: A Developer's Guide, 2nd Edition, by Cary Jensen and Loy Anderson (2010, ISBN: 1453769978). For information on this book and on purchasing this book in various formats (print, e-book, etc), visit: http://www.JensenDataSystems.com/ADSBook10

Without a doubt, Delphi developers have the richest choice of options for accessing Advantage. In addition to being able to use ACE (Advantage Client Engine), Delphi developers can use the Advantage ODBC Driver, the Advantage OLE DB Provider, and the Advantage .NET Data Provider (with Delphi 2005 through Delphi 2007, as well as with Delphi Prism). But there is one data access mechanism that beats them all—the Advantage Delphi Components. Using the Advantage Delphi Component classes, Delphi developers can utilize almost every feature available in Advantage. Only ACE provides access to more capabilities, but at the expense of being significantly more complicated to use.

Note: The Advantage Delphi Components are also used with the later versions C++Builder. If you are using Delphi version 7 or earlier, or C++Builder versions 5-7, install the Advantage TDataSet Descendants. You use the Advantage TDataSet Descendants in the same way as you use the Advantage Delphi Components.

This chapter provides you with examples of using the Advantage Delphi Component classes to perform a wide range of common data-related tasks using Delphi. As is the case with the other chapters in Part III, this discussion assumes that you are already familiar with the development environment being described. Instead, the focus is on code that works with the data dictionary you have been using throughout this book.

Note: If you want to build .NET applications using Delphi 2005, Delphi 2006, Delphi 2007, or Delphi Prism, see the descriptions in Chapter 17, Advantage and the .NET Data Provider.

Advantage and Delphi

Delphi has been the development environment of choice for Advantage developers for some time. While we anticipate that other languages, such as C#, will be the choice of a growing number of Advantage developers due to the availability of the Advantage .NET Data Provider, Delphi continues to be an excellent tool for working with Advantage.

There are two primary reasons why Advantage and Delphi go so well together. The first is the abundance of Advantage features that are available through the Advantage Delphi Component classes. Using these components, it is extremely easy to access, change, and manage your Advantage data.

The second reason is that the data access model supported by the BDE (Borland Database Engine), and encapsulated in the TDataSet abstract class of the VCL (Visual Component Library), fits beautifully with the navigational model that Advantage's ISAM architecture permits. This reason is actually closely related to the first. The ease with which Delphi developers can use Advantage is a direct result of the nearly seamless way in which the TDataSet interface and the Advantage API (application programming interface) interact.

But there is more. The Advantage components that descend from TDataSet do much more than simply conform to the TDataSet interface. They also expose a large number of functions in the ACE API. For example, you can call AdsStmtSetTablePassword on an AdsQuery to submit a free table's encryption password before attempting to query it. This method is not part of the TDataSet interface, but is exposed by this component to simplify your work with ADS without having to deal directly with ACE.

An interesting piece of evidence for this near perfect fit between Advantage and Delphi is that the Advantage Data Architect is written in Delphi. In fact, when you install the Advantage Data Architect, most of the Delphi source files and packages that are used to build the Advantage Data Architect are included. These Delphi units can be a fascinating source of ideas if you need to perform some task that is out of the ordinary. (The Advantage Data Architect source files do not include some third-party components that were used to build the Advantage Data Architect. As a result, you cannot actually use these source files to recompile the Advantage Data Architect unless you also have licenses for these third-party components.)

That the Advantage Data Architect was written in Delphi presented a creative opportunity for integration between Advantage and Delphi. While working on the Advantage Data Architect, Advantage R&D Product Manager J.D. Mullin realized that he could create property editors for the TAdsTable and TAdsQuery components in Delphi that surface the Advantage Data Architect's Table Designer (for TAdsTable) and the SQL Utility (for TAdsQuery).

As a result, you can right-click a configured TAdsTable in Delphi and select ALTER/Restructure Table. Doing so opens the Table Designer for the associated table, permitting you to change the table's structure, modify table properties, and configure table indexes—all without having to load a separate copy of the Advantage Data Architect.

Similarly, you can click the ellipsis button for the SQL property of an AdsQuery to open the SQL Utility. Here you can test, debug, and perfect your SQL.

Finally, there is a new component that ships with the Advantage Delphi Components for Advantage 10. The TAdsEvent component makes receiving notifications from client applications almost effortless. This component takes responsibility for creating a worker thread from which it waits to receive notifications, and exposes an event handler that it will call, and which will execute in the primary thread of execution, when a notification is received. (For a detailed discussion of the role of notifications and worker threads, see Chapter 9, *Using Notifications*).

Understanding the Delphi Components

When you install the Delphi Components, a new page is inserted into the Tool Palette. The components installed on this page include TAdsConnection, TAdsTable, TAdsQuery, and TAdsStoredProc. You also get the TAdsBatchMove, TAdsDictionary, TAdsSettings, and TAdsEvent classes installed as well. These components are shown in Figure 17-1.

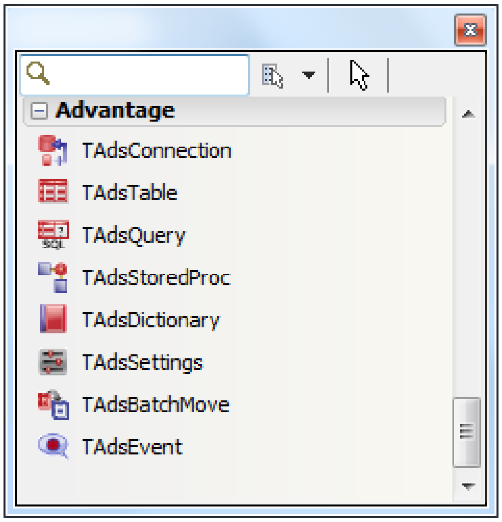


Figure 17-1: The Advantage page of the Tool Palette in Delphi

That TAdsTable, TAdsQuery, and TAdsStoredProc all descend from TDataSet means that they share a common programming interface with all other standard Delphi data access components. This interface is largely navigational, although it also supports the entire range of set-based operations of Advantage SQL through the TAdsQuery class.

Because the Advantage TDataSet descendants share a common interface with other Delphi data access components, such as TTable, TSQLDataSet, TIBQuery, TClientDataSet, among others, converting an existing Delphi application to use Advantage is pretty straightforward. In fact, now that Borland has officially deprecated Borland SQL Links for Windows (the portion of the BDE based on DLLs that provided the BDE with access to many remote database servers), Advantage is an ideal option when choosing an alternative data access mechanism, especially if your application already uses the navigational model. (Migrating Delphi BDE applications to SQL-based remote database servers almost always involves a major update to the user interface, in order to accommodate the limitations of set-based SQL.)

The remainder of this chapter shows you how to access Advantage using Delphi. This discussion is divided into three parts. The first part describes common basic tasks, such as connecting to Advantage and accessing data. The second part shows you how to leverage the navigational model with Delphi and Advantage. The third part demonstrates several basic administrative tasks, such as creating tables and granting rights to them.

Code Download: The Delphi project Delphi_ADC.dpr can be found with this book's code download (see Appendix A). This project was created and tested using Delphi 2010, but is designed to be compatible with Delphi versions 5 and later.

All of the examples presented here can be found in a Delphi project named Delphi_ADC. Figure 17-2 shows the main form of this project in Delphi 2010.

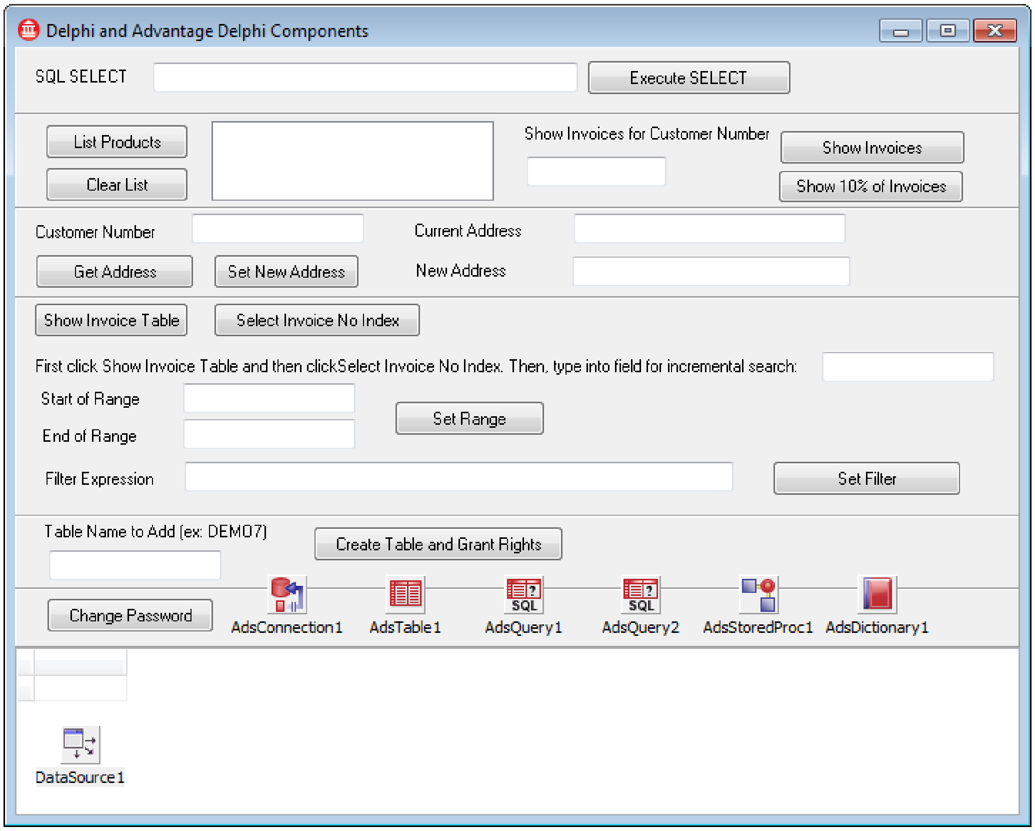


Figure 17-2: The Delphi_ADC project's main form

A final comment about this project is in order. Many of the operations performed in this project make use of the TAdsTable component, which is pretty common for Delphi applications. Many of these same operations can be performed using a SQL query executed through an AdsQuery or by calling stored procedures using the TAdsStoredProc component (and often more efficiently, as well). For examples of possible SQL alternatives, refer to Chapter 18, and Chapters 20 through 23.

Performing Basic Tasks with Advantage and Delphi

This section describes some of the more common tasks that you can perform with Delphi. These include connecting to a data dictionary, opening a table, executing a query, using a parameterized query, and executing a stored procedure.

Connecting to Data

You connect to a data dictionary or a directory in which free tables are located using a TAdsConnection component. TAdsConnection is very similar to the VCL's TDatabase component, located on the BDE page of the Tool Palette, and the TSQLConnection component, located on the dbExpress page of the Tool Palette. At a minimum, you must set a property of this TAdsConnection component to describe where the data dictionary or free tables are located, after which you set the IsConnected property to True (or call the Connect method). If you are connecting to a data dictionary that requires a login, and you do not want the user to have to enter their user name, you must also set the Username and Password properties of this component.

There are two ways to indicate the location of your data. The easiest way, if you have a connection defined for your data dictionary or free table directory, is to set the AliasName property of the TAdsConnection to a defined connection. Connections are defined in the ads.ini file, which can be stored in the same directory as your client application (under Windows), or anywhere on your search path. (When you use the Advantage Data Architect, the ads.ini file is stored in the Windows directory by default.)

Alternatively, you can set the ConnectPath of the TAdsConnection to a path, preferably a UNC path (which can include an IP address and port number if Advantage is installed on a machine with a fixed IP address). The benefit of using an ads.ini file is that you can change the directory in which Advantage looks for your data by changing an entry in the ads.ini file. By comparison, if your data directory is hard-coded in your application, changing your data directory involves recompiling and redistributing your client application.

Tip: Some developers who do not want to use the ads.ini file assign the ConnectPath property of the TAdsConnection object at runtime. An alternative is to store the connection path in an external file that you read manually, using a suitable class in Delphi, such as TStringList or TInitFile.

If you set the IsConnected property of the TAdsConnection to True at design time, and a user name and password are supplied (if necessary), you will have an active connection within the Delphi IDE, and the connection will be reestablished at runtime once the

TAdsConnection is created and initialized (so long as the StoredConnected property is set to True, which is the default). If IsConnected is set to False at design time, the TAdsConnection component will establish a connection at runtime once either the IsConnected property is explicitly set to True, Connect is invoked, or any TDataSets that use the TAdsConnection attempt to open or execute.

If you set the Username and Password properties prior to connecting a TAdsConnection, set the LoginPrompt property to False. If LoginPrompt is True, the user will be prompted for their user name and password.

Tip: If LoginPrompt is set to True, but no login dialog box is displayed, you must add the DBLogDlg unit to your unit's uses clause.

In the Delphi_ADC project, the Alias property of the TAdsConnection is set to DemoDictionary (the connection name you defined for the DemoDictionary data dictionary in Chapter 4, *Understanding and Using Data Dictionaries*), the user name is set to **adsuser**, the password is set to **password**, and the LoginPrompt is set to False. With these settings in place, you will not be prompted for a user name or password when connecting to the data dictionary.

Note: If you have difficulty connecting, it might be because you have other client applications, such as the Advantage Data Architect, connected using a local connection. Ensure that all clients connecting to the database use the same type of connection. You control the type of connections attempted by a TAdsConnection using the AdsServerTypes property.

Accessing an ADS Table

The easiest way to access data is with a TAdsTable component. At a minimum, you must set the TAdsTable's AdsConnection property to a TAdsConnection (this is often done at design time), its TableName property to the name of a table or view you want to access, and the Active property to True. Calling the TAdsTable's Open method sets the TAdsTable's Active property to True.

This is demonstrated in the following event handler, which is associated with the Show Invoice Table button (shown in Figure 17-2):

```
procedure TForm1.ShowInvoiceBtnClick(Sender: TObject);
begin
  if AdsTable1.Active then AdsTable1.Close;
  AdsTable1.TableName := 'INVOICE';
  AdsTable1.Open;
  DataSource1.DataSet := AdsTable1;
end;
```

Reading and Writing Data

You access individual records in a TDataSet using its Fields property or its FieldByName method. Fields is a collection property, and you must pass it an index that identifies which field you want to read from or write to, based on its zero-based ordinal position in the table's structure. When you invoke FieldByName, you pass a string containing the name of the field that you want to work with. This approach works for any TAdsTable, TAdsQuery, or TAdsStoredProc component that returns a result set.

The following event handler, associated with the Get Address button (shown in Figure 17-2), demonstrates how to read a field:

```
procedure TForm1.GetAddressBtnClick(Sender: TObject);
begin
  if AdsTable1.Active then AdsTable1.Close;
  AdsTable1.TableName := 'CUSTOMER';
  AdsTable1.IndexName := 'Customer ID';
  AdsTable1.Open;
  if AdsTable1.FindKey([CustNoText.Text]) then
    OldAddressText.Text :=
      AdsTable1.FieldByName('Address').AsString
  else
    ShowMessage('Customer ID ' + CustNoText.Text +
      ' not found');
  DataSource1.DataSet := AdsTable1;
end;
```

The following event handler, associated with the Set New Address button (shown in Figure 17-2), demonstrates writing to a field:

```
procedure TForm1.SetAddressBtnClick(Sender: TObject);
begin
  if AdsTable1.Active then AdsTable1.Close;
  AdsTable1.TableName := 'CUSTOMER';
  AdsTable1.IndexName := 'Customer ID';
  AdsTable1.Open;
  if AdsTable1.FindKey([CustNoText.Text]) then
  begin
    AdsTable1.Edit;
    AdsTable1.FieldByName('Address').AsString :=
      NewAddressText.Text;
    AdsTable1.Post;
  end
  else
    ShowMessage('Customer ID ' + CustNoText.Text +
      ' not found');
  DataSource1.DataSet := AdsTable1;
end;
```

Executing a Query

You define a query by assigning the SQL statement you want to execute to the SQL StringList property of a TAdsQuery that is associated with a TAdsConnection through its AdsConnection property. If the query returns a result set, you execute it by calling its

Open method or by setting its Active property to True. If the query does not return a result set, execute it by calling its ExecSQL method.

Note: The SQL that you can assign to the SQL property of a TAdsQuery can be either a single SQL statement or an Advantage SQL script. Advantage SQL scripts are described in Chapter 12, Introduction to Using Advantage SQL .

By default, you cannot edit the result set returned by a TAdsQuery. If your query produces a live cursor, setting the TAdsQuery's RequestLive property to True permits you to edit the data in the result set.

The following code demonstrates the execution of a query entered by the user. It is associated with the Execute SELECT button (shown in Figure 17-2):

```
procedure TForm1.DoSelectClick(Sender: TObject);
begin
  if AdsQuery1.Active then AdsQuery1.Close;
  AdsQuery1.SQL.Text := SELECTText.Text;
  AdsQuery1.Open;
  DataSource1.DataSet := AdsQuery1;
end;
```

Using a Parameterized Query

The TAdsQuery component supports both named and positional parameters. You must bind data to every parameter of a parameterized query prior to executing it. You can do this either using the Params property, which is a collection property indexed by parameter position, or the ParamByName method, which takes a parameter name as an argument. Both of these approaches return a TParam, which you use to assign a value to the parameter.

The AdsQuery2 object on this project's main form has the following SQL statement assigned to its SQL property:

```
SELECT * FROM INVOICE WHERE [Customer ID] = :cust
```

The parameter, named cust, is bound, and the query executed, from the event handler shown in the following code segment. This event handler is associated with the Show Invoices button (shown in Figure 17-2):

```
procedure TForm1.ShowInvoicesBtnClick(Sender: TObject);
begin
  if AdsQuery2.Active then AdsQuery2.Close;
  if ParamText.Text = '' then
  begin
    ShowMessage('Please enter a customer number');
    Exit;
  end;
  AdsQuery2.Params[0].AsInteger := StrToInt(ParamText.Text);
  AdsQuery2.Open;
  DataSource1.DataSet := AdsQuery2;
end;
```

Calling a Stored Procedure

Stored procedures are invoked using the EXECUTE PROCEDURE SQL statements in most of the Advantage data access mechanisms, and you can use a TAdsQuery in Delphi to do the same. But Delphi developers have an alternative solution—being able to invoke a stored procedure using the TAdsStoredProc component.

Note: When using a TAdsQuery in Advantage 10, you can invoke a stored procedure either using the EXECUTE PROCEDURE statement, or you can include the stored procedure invocation in the FROM clause of a SELECT query, as described in Chapter 12, Introduction to Using Advantage SQL.

There are several advantages to invoking a stored procedure using a TAdsStoredProc. The first is that you can use the Params property of the stored procedure to configure and assign the stored procedure's input parameters. After configuring a stored procedure in the Delphi IDE, you can select the Params property editor of the stored procedure to view the names and data types of each parameter. The names and data types of each of the stored procedure's parameters automatically appear in the Params property editor. If you like, you can then assign default values to the parameters that appear in the Params property editor, though binding values to parameters is often something that you do at runtime.

Note: If the stored procedure invocation in a TAdsQuery includes parameters (named or positional), these also appear in the Params property of the TAdsQuery component automatically.

Another advantage of TAdsStoredProc is realized when the stored procedure returns a result set. Specifically, a stored procedure that returns a result set can be treated exactly the same as a TAdsTable or a TAdsQuery that returns a result set. Specifically, the TAdsStoredProc can be assigned to the DataSet property of a TDataSource so that the returned data can be associated with data-aware controls. The TAdsStoredProc also populates any output parameters, which can be read individually using the Params property or ParamByName method.

At a minimum, you must set the AdsConnection and the StoredProcName properties of the stored procedure. Also, you must assign values to all input parameters, if present, prior to executing the stored procedure. Set the TAdsStoredProc component's ParamBindMode property to pbByName or pbByNumber, based on whether you want to bind parameters by name or position, respectively (the default value is pbByName).

If your stored procedure returns a result set, you execute it by calling its Open method, or by setting its Active property to True. If your stored procedure does not return a result set, you execute it by calling its ExecProc method.

If you want to execute a given stored procedure more than once, and with different values passed to its input parameters, you must first close the stored procedure before changing any input parameters. However, this is not necessary for a stored procedure that does not return a result set.

The use of a stored procedure is demonstrated by the following code associated with the OnClick event handler for the Show 10% of Invoices button (shown in Figure 17-2). This code assumes that the SQL stored procedure created in Chapter 7, Get10PercentSQL, is assigned to the StoredProc property of the TAdsStoredProc component referenced in this code:

```
procedure TForm1.CallStoredProcBtnClick(Sender: TObject);
begin
  if AdsStoredProc1.Active then AdsStoredProc1.Close;
  if ParamText.Text = '' then
  begin
    ShowMessage('Please enter a customer number');
    Exit;
  end;
  AdsStoredProc1.Params[0].Value := ParamText.Text;
  try
    AdsStoredProc1.Open;
  except
    on e: Exception do
      ShowMessage(e.Message);
  end;
  DataSource1.DataSet := AdsStoredProc1;
end;
```

To view the stored procedure's parameters, select AdsStoredProc1 on the main form and then, using the Object Inspector, select the Params property and click the ellipsis button that appears. Delphi displays the parameters in the Params collection editor shown in Figure 17-3.

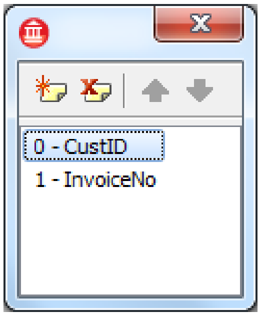
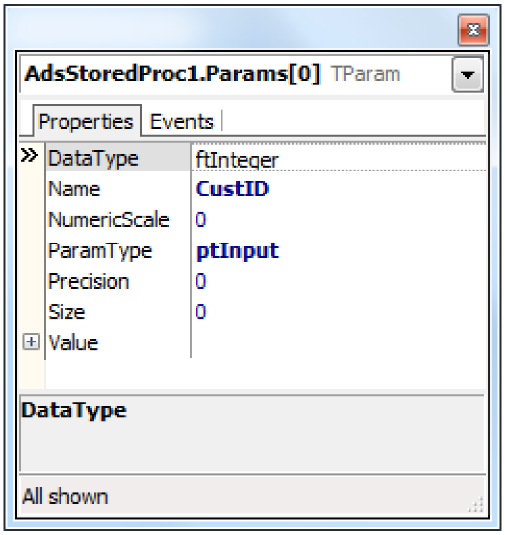


Figure 17-3: The Params collection editor

If you select one of the available parameters in the Params collection editor, you can view and edit the parameter's properties using the Object Inspector shown in Figure 17-4.



DataType	ftInteger
Name	CustID
NumericScale	0
ParamType	ptInput
Precision	0
Size	0
Value	

DataType

Figure 17-4: The Object Inspector

Navigational Actions with Advantage and Delphi

The TDataSet class of the VCL supports a wide range of navigational actions. While all of these actions apply to TAdsTables, many of them are also supported by the TAdsQuery and TAdsStoredProc components when they return a result set.

Setting an Index

If you are connected to a data dictionary, and you have designated a default index for a table, that index will automatically be used when you open that table using a TAdsTable. If you do not have a default index, or want to switch to some other index, you set the IndexName property of the TAdsTable to the name of the index you want to use. This is demonstrated in the following code segment, which is associated with the Select Invoice No Index button (shown in Figure 17-2):

```
procedure TForm1.SetIndexBtnClick(Sender: TObject);
begin
  if (AdsTable1.TableName <> 'INVOICE') and
    (AdsTable1.Active) then
    AdsTable1.Close;
  AdsTable1.IndexName := 'Invoice No';
end;
```

Finding a Record Based on Data

TDataSets support a number of methods for locating records based on their contents. These include FindKey, FindNearest, and Locate, in addition to the brute-force method of scanning through every record in the table. Of these techniques, FindKey and FindNearest are the fastest and easiest to use for locating a record using the currently selected index.

Both of these methods take a single parameter consisting of a constant array. You represent a constant array by enclosing one or more values within parentheses, separating the values using commas when there is more than one value. Both of these methods search for the first element of the array in the first field of the current index, then search for the second value, if present, in the second field of the current index, and so on.

FindKey is a Boolean function that returns True if a matching record is found. If FindKey returns True, the located record is made the current record. FindNearest is a procedure. It always repositions the current record to the closest match to the search criteria (which might actually be the current record).

The use of FindNearest is demonstrated in the following event handler. This event handler is associated with the OnChange event of the TEdit named SearchText. After clicking the Show Invoice Table and the Select Invoice No Index buttons, this code permits an incremental search through the INVOICE table:

```
procedure TForm1.SearchTextChange(Sender: TObject);
begin
  if (not AdsTable1.Active) or
    (AdsTable1.TableName <> 'INVOICE') or
    (AdsTable1.IndexName <> 'Invoice No') then
    begin
      ShowMessage('Click Show Invoice Table and Select ' +
        'Invoice No Index before performing an ' +
        'incremental search');
      Exit;
    end;
  AdsTable1.FindNearest([SearchText.Text]);
end;
```

Note: Unlike the BDE, ADS employs indexes with the Locate and Lookup methods whenever appropriate indexes are available, and creates AOFs (Advantage Optimized Filters) when they are not available. As a result, these methods are much faster than their BDE counterparts. However, Locate and Lookup are more complicated to use than FindKey and FindNearest. For information on using Locate and Lookup, refer to the Advantage help.

Setting a Range

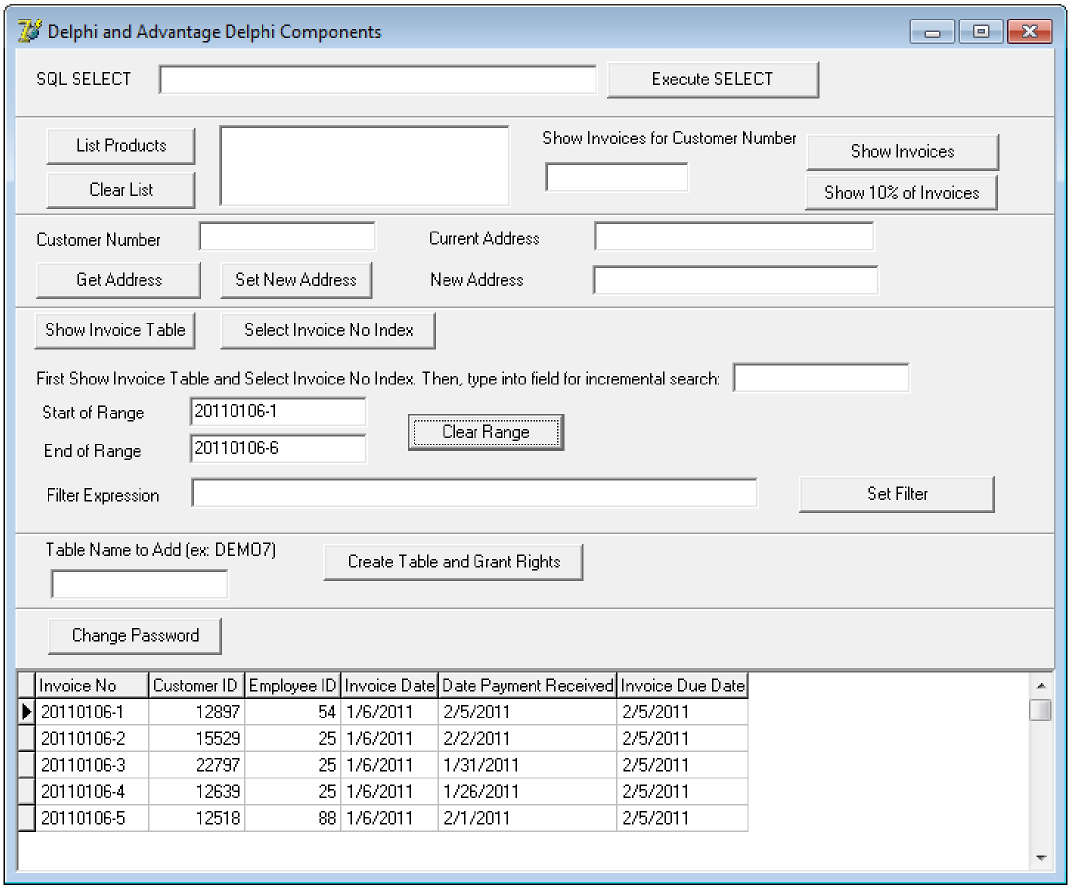
Delphi uses the term *range* instead of *scope*, but it means the same thing. A range defines a subset of records to view in a table, based on an index.

You set a range in Delphi by calling SetRange. This method takes two parameters, both of which are constant arrays. The first constant array contains the beginning values of the range, where the first value defines the beginning of the range based on the first field of the current index, the second value, if present, defines the beginning of the range on the second field of the index, and so on. The second constant array contains the ending values of the range, again on a field-by-field basis, based on the current index.

The following code, which is associated with the OnClick event handler of the Set Range button, demonstrates how to apply a range:

```
procedure TForm1.SetRangeBtnClick(Sender: TObject);
begin
  if SetRangeBtn.Caption = 'Set Range' then
  begin
    if (not AdsTable1.Active) or
      (AdsTable1.TableName <> 'INVOICE') or
      (AdsTable1.IndexName <> 'Invoice No') then
      begin
        ShowMessage('Click Show Invoice Table ' +
          'and Select Invoice '+
          'No Index before setting a range');
        Exit;
      end;
    AdsTable1.SetRange([StartRange.Text], [EndRange.Text]);
    SetRangeBtn.Caption := 'Clear Range';
  end
  else
  begin
    AdsTable1.CancelRange;
    SetRangeBtn.Caption := 'Set Range';
  end;
end;
```

An example of a range set on the INVOICE table is shown in Figure 17-5. Notice that there are only five records, out of more than 2,000 records, in this range (there is no record for 20110106-6).



Start of Range: 20110106-1
End of Range: 20110106-6

Invoice No	Customer ID	Employee ID	Invoice Date	Date Payment Received	Invoice Due Date
20110106-1	12897	54	1/6/2011	2/5/2011	2/5/2011
20110106-2	15529	25	1/6/2011	2/2/2011	2/5/2011
20110106-3	22797	25	1/6/2011	1/31/2011	2/5/2011
20110106-4	12639	25	1/6/2011	1/26/2011	2/5/2011
20110106-5	12518	88	1/6/2011	2/1/2011	2/5/2011

Figure 17-5: A range limits the records available in a table

***NOTE:** If you are testing this code and you set a range, clear the range before continuing to the next section. Clear the range by clicking the Clear Range button (the button whose caption alternates between Set Range and Clear Range).*

Setting a Filter

A filter is similar to a range in that it can limit a table to a subset of records. Unlike a range, however, a filter does not rely on the current index.

You set a filter by assigning a filter expression to a TAdsTable's Filter property, and then setting the Filtered property to True. You drop a filter by setting the Filter property to an empty string or by setting Filtered to False (or both). Setting and dropping a filter is demonstrated in the following code, which is located in the OnClick event handler for the Set Filter button (shown in Figure 17-2):

```
procedure TForm1.FilterBtnClick(Sender: TObject);
begin
```

```
if FilterBtn.Caption = 'Drop Filter' then
  begin
    AdsTable1.Filtered := False;
    FilterBtn.Caption := 'Set Filter';
  end
  else
  begin
    if (not AdsTable1.Active) or
      ( not (DataSource1.DataSet is TAdsTable)) then
      begin
        ShowMessage('Please open a table before '+
          'setting a filter');
        Exit;
      end;
    AdsTable1.Filter := FilterText.Text;
    AdsTable1.Filtered := True;
    FilterBtn.Caption := 'Drop Filter';
  end;
end;
```

If you run this project and click the Show Invoice Table button, and then enter the following filter expression, the DBGrid at the bottom of the form will display only two records if you click the Set Filter button:

```
Customer ID = 12037 and Employee ID = 89
```

Note: Although a filter does not rely on the current index, the speed with which a filter can be applied is directly related to the available indexes on the table. Specifically, filters use AOFs (Advantage Optimized Filters). AOFs are described in Chapter 3, Defining Indexes. As a result, filters can be applied quickly when the expressions in the filter expression map to available indexes on the underlying table. By comparison, filters applied when you use the BDE do not use indexes, making filters under Advantage significantly faster than the corresponding filters under the BDE.

Note: If you are testing this code, and you set a filter, drop the filter before continuing to the next section. Drop the filter by clicking the Drop Filter button (the button whose caption alternates between Set Filter and Drop Filter).

Scanning a Result Set

Scanning is the process of sequentially reading every record in a result set or table, or every record in the range and/or filtered view of the result set or table if either a filter or a range, or both, are active. In most cases, scanning involves an initial navigation to the first record of the result set or table, followed by repeated calls to advance one record until all of the records have been visited (or until you find a record for which you are scanning).

Although scanning is a common task, it is important to note that it necessarily requires the client application to retrieve all of the records in the result set. This is not a problem when few records are involved, but if a large number of records are being scanned, network resources may be taxed.

Tip: If you are using ADS, and you must scan a large number of records, implement the operation using a stored procedure as described in Chapter 7, Creating Stored Procedures. Scanning from a stored procedure executed from ADS requires no network resources when the server and the data are located on the same machine.

The following code demonstrates the scanning of a TAdsTable. This code, associated with the OnClick event handler of the List Products button (shown in Figure 17-2), navigates the entire PRODUCTS table, assigning data from each record to the ProductList list box:

```
procedure TForm1.ListProductsBtnClick(Sender: TObject);
begin
  DataSource1.DataSet := nil;
  if AdsTable1.Active then AdsTable1.Close;
  AdsTable1.TableName := 'PRODUCTS';
  AdsTable1.Open;
  AdsTable1.First;
  while not AdsTable1.EOF do
  begin
    ProductList.Items.Add(AdsTable1.Fields[0].AsString +
      '       ' + AdsTable1.Fields[1].AsString);
    AdsTable1.Next;
  end;
  AdsTable1.Close;
end;
```

Administrative Operations with Advantage and Delphi

While Advantage requires little in the way of periodic maintenance to keep it running smoothly, many applications need to provide administrative functionality related to the management of users, groups, and objects.

This section is designed to provide you with insight into exposing administrative functions in your client applications. Two related, yet different, operations are demonstrated here. In the first, a new table is added to the database and all groups are granted access rights to it. This operation requires that you establish an administrative connection (or a user connection that has the appropriate WITH GRANT privileges).

The second operation involves permitting individual users to modify their own passwords. Especially in the security-conscious world of modern database management, this feature is often considered an essential step to protecting data.

Like many of the other operations described in this chapter, Delphi provides a variety of different means for implementing these features. An obvious solution is to use SQL queries to perform these tasks. This approach is demonstrated in other chapters in this part of the book, and in many of those cases represents the only mechanism available.

Since the SQL approach is shown elsewhere in Chapters 18 and Chapters 20 through 23, the following sections demonstrate how to implement these operations using methods of the TAdsTable, TAdsConnection, and TAdsDictionary components. While this approach is sometimes more involved than the SQL approach, it is nonetheless valuable in that it demonstrates the utility of the various Advantage Delphi Components components. If you prefer to use the SQL approach in your Delphi applications, refer to the SQL statements used the other chapters.

Note: Although the TAdsDictionary does enable you to perform the tasks we describe in this section, and we sincerely believe that it is interesting to know how this interface differs from the SQL-based interface, there is an additional reason to consider the SQL versions of these operations as presented in the other chapters of this part of the book. The TAdsDictionary is deprecated. Some of the newest administrative operations in Advantage cannot be performed with TAdsDictionary. In order to perform these tasks, you will have to use SQL.

Creating a Table and Granting Rights to It

The Delphi_ADC project permits a user to enter the name of a table that will be created in the data dictionary, after which all non-default groups will be granted rights to the table. This operation is demonstrated in the following event handler, which is associated with the OnClick event of the Create Table and Grant Rights button (shown in Figure 17-2). Unlike most of the code shown in this chapter, several comment lines are retained here, due to its complexity:

```
procedure TForm1.CreateTableBtnClick(Sender: TObject);
var
  AdminConnection: TAdsConnection;
  AdminTable: TAdsTable;
  StringList: TStringList;
  i: Integer;
begin
  if TableNameText.Text = '' then
  begin
    ShowMessage('Please enter the name of the ' + 'table to create');
    Exit;
  end;
  // Create Connection and Table objects
  AdminConnection := TAdsConnection.Create(nil);
  AdminConnection.Name := 'Admin';
  AdminTable := TAdsTable.Create(AdminConnection);
  AdminTable.DatabaseName := AdminConnection.Name;
  StringList := TStringList.Create;
  try
    // Configure Connection and Table objects
    AdminConnection.AliasName := AdsConnection1.AliasName;
    AdminConnection.AdsServerTypes := AdsConnection1.AdsServerTypes;
    AdminConnection.Username := 'adssys';
    // Add the password if ADSSYS uses a password
    // AdminConnection.Password := 'password';
    AdminConnection.LoginPrompt := False;
    AdminConnection.IsConnected := True;
    AdminConnection.GetTableNames(StringList, TableNameText.Text);
    for i := 0 to Pred(StringList.Count) do
      if AnsiCompareText(StringList.Strings[i],
        TableNameText.Text) = 0 then
      begin
        ShowMessage('This table already exists. ' + 'Cannot create');
        Exit;
      end;
    // Define new table structure and create it
    with AdminTable.FieldDefs do
    begin
      Add('Full Name', ftString, 30);
      Add('Date of Birth', ftDate);
      with AddFieldDef do
      begin
        Name := 'Credit Limit';
        DataType := ftBCD;
        Precision := 20;
        Size := 4;
      end;
      Add('Active', ftBoolean);
    end;
    AdminTable.TableType := ttAdsADT;
    AdminTable.TableName := TableNameText.Text;
    AdminTable.CreateTable;
    // Configure and connect the AdsDictionary object
    AdsDictionary1.AliasName := AdminConnection.AliasName;
    AdsDictionary1.AdsServerTypes := AdminConnection.AdsServerTypes;
    AdsDictionary1.Username := AdminConnection.Username;
    AdsDictionary1.Password := AdminConnection.Password;
    AdsDictionary1.LoginPrompt := False;
    AdsDictionary1.Connect;
    StringList.Clear;
    AdsDictionary1.GetGroupNames(StringList);
    if StringList.Count = 0 then
    begin
      ShowMessage('No groups to grant rights to');
      Exit;
    end;
    // Grant access rights to all groups
    for i := 0 to Pred(StringList.Count) do
      if Pos('DB:', StringList.Strings[i]) = 0 then
          AdsDictionary1.SetObjectAccessRights(TableNameText.Text,
            StringList.Strings[i], 'RW');
  finally
    // cleanup
    AdsDictionary1.Disconnect;
    StringList.Free;
    AdminTable.Free;
    AdminConnection.IsConnected := False;
    AdminConnection.Free;
  end;
  ShowMessage('The ' + TableNameText.Text + ' table has been ' +
      'created, with rights granted to all groups');
end;
```

This event handler demonstrates a number of interesting techniques. First, while it uses a TAdsDictionary component that was placed at design time onto the main form (shown in Figure 17-2), the TAdsConnection and TAdsTable used by the administrative connection are created and then discarded at runtime. This approach is always valid, and could have been used by many of the other event handlers listed in this chapter. In most cases, however, components that are placed and configured at design time are easier to maintain.

After verifying that the requested table does not already exist in the data dictionary, the administrative connection is configured and opened, and a new TAdsTable is configured to use it. Next, the table's structure is defined (using both the Add and AddFieldDefs methods of the TAdsTable), and the table is created with a call to CreateTable.

Note: The administrative user name and passwords are represented by string literals in this code segment. This was done for convenience, but in a real application, either you would ask for this information from the user or you would encrypt this data to prevent its discovery. (String literals can be inspected in compiled applications using a variety of tools.)

After creating the table, the TAdsDictionary component is configured and opened. Finally, the list of groups is retrieved and used to grant read and write access to the table.

Changing a User Password

A user can change the password on the user’s own connection, if you permit this change. In most cases, only when every user has a distinct user name would you expose this functionality in a client application. When multiple users share a user name, this operation is usually reserved for an application administrator.

As was done in the preceding section, this code demonstrates changing a user password using a TAdsDictionary component. For an example of changing a password using the sp_ModifyUserProperty stored procedure, refer to Chapter 18, and Chapters 20 through 23.

```
procedure TForm1.ChangePasswordBtnClick(Sender: TObject);
var
  Username: String;
  OldPass: String;
  NewPass1: AnsiString;
  NewPass2: AnsiString;
{$HINTS OFF}
  function CheckPass(UName, UPass: String): Boolean;
  var
    TempConnection: TAdsConnection;
  begin
```

```
    result := False;
    TempConnection := TAdsConnection.Create(nil);
    try
      TempConnection.AliasName := AdsConnection1.AliasName;
      TempConnection.AdsServerTypes := AdsConnection1.AdsServerTypes;
      TempConnection.Username := UName;
      TempConnection.Password := UPass;
      TempConnection.LoginPrompt := False;
      try
        TempConnection.IsConnected := True;
      except
        result := False;
      end;
      result := True;
    finally
      TempConnection.IsConnected := False;
      TempConnection.Free;
    end;
  end; // CheckPass
{$HINTS ON}

begin
  Username := AdsConnection1.Username;
  OldPass := 'Enter password';
  OldPass := InputBox('Password',
    'Enter your current password', OldPass);
  if OldPass = 'Enter password' then
    Exit;

  if not CheckPass(Username, OldPass) then
  begin
    ShowMessage('Cannot validate your current password. ' +
        'Cannot change password');
    Exit;
  end;

  NewPass1 := '';
  NewPass2 := '';
  NewPass1 := InputBox('Password', 'Enter your new Password', NewPass1);
  if NewPass1 = '' then
  begin
    ShowMessage('Password cannot be blank. ' +
      'Cannot change password');
    Exit;
  end;
  NewPass2 := InputBox('Password',
    'Confirm your new password', NewPass2);
  if NewPass1 <> NewPass2 then
  begin
    ShowMessage('Passwords did not match. ' + 'Cannot change password');
    Exit;
  end;

  // Connect AdsDictionary1 and change password
  AdsDictionary1.AliasName := AdsConnection1.AliasName;
  AdsDictionary1.AdsServerTypes := AdsConnection1.AdsServerTypes;
  AdsDictionary1.Username := AdsConnection1.Username;
  AdsDictionary1.Password := OldPass;
  AdsDictionary1.LoginPrompt := False;
  AdsDictionary1.Connect;
  AdsDictionary1.SetUserProperty(AdsConnection1.Username,
    ADS_DD_USER_PASSWORD, PAnsiChar(NewPass1),
    StrLen(PAnsiChar(NewPass1)) + 1);
  AdsDictionary1.Disconnect;

  ShowMessage('Password successfully changed. ' +
      'New password will be valid next time you connect');
end;
```

This code segment is a bit simpler than the preceding one. After verifying that the user knows the current password with which they are connected, they are prompted for their new password twice, for confirmation purposes. Next, the TAdsDictionary component is configured to connect with the user account, after which its SetUserProperty method is invoked to change the user's password. As the final dialog box displayed by this event handler indicates, this password will be valid once the user terminates all connections on this user account.

Note: If you run this code, and change the password of the adsuser account, you should use the Advantage Data Architect (or this sample client application) to change the password back to ***password****. Otherwise, you will not be able to run this project again, since the literal 'password' is assigned to the Password property of the TAdsConnection component in this project.*