# Developer's Guide

Part II

## Developing Database Applications

Bookmark links added by Michael J. Riley
http://capecodgunny.blogspot.com

Borland®
Delphi™ 7
for Windows®

Borland Software Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249
www.borland.com

Part II

# Developing database applications

Chapter 19

## Designing database applications

Chapter 20

## Using data controls

Chapter 20

# Using data controls (continued)

Chapter 21

# Creating reports with Rave Reports

Chapter 21
# Creating reports with Rave Reports (continued)

Chapter 22
# Using decision support components

Chapter 24

# Understanding datasets (continued)

Chapter 24

# Understanding datasets (continued)

Chapter 25

# Working with field components

Chapter 25

# Working with field components (continued)

Chapter 26

# Using the Borland Database Engine

Chapter 26
# Using the Borland Database Engine (continued)

Chapter 27
# Working with ADO components

Chapter 27
# Working with ADO components (continued)

Chapter 28
# Using unidirectional datasets

Chapter 29

# Using client datasets

Chapter 29

# Using client datasets (continued)

Chapter 30

# Using provider components

Chapter 31

# Creating multi-tiered applications

Chapter 31

# Creating multi-tiered applications (continued)

Chapter 32

# Using XML in database applications

Chapter

19

# Designing database applications

Database applications let users interact with information that is stored in databases. Databases provide structure for the information, and allow it to be shared among different applications.

Delphi provides support for relational database applications. Relational databases organize information into tables, which contain rows (records) and columns (fields). These tables can be manipulated by simple operations known as the relational calculus.

When designing a database application, you must understand how the data is structured. Based on that structure, you can then design a user interface to display data to the user and allow the user to enter new information or modify existing data.

This chapter introduces some common considerations for designing a database application and the decisions involved in designing a user interface.

## Using databases

Delphi includes many components for accessing databases and representing the information they contain. They are grouped according to the data access mechanism:

- The BDE page of the Component palette contains components that use the Borland Database Engine (BDE). The BDE defines a large API for interacting with databases. Of all the data access mechanisms, the BDE supports the broadest range of functions and comes with the most supporting utilities. It is the best way to work with data in Paradox or dBASE tables. However, it is also the most complicated mechanism to deploy. For more information about using the BDE components, see Chapter 26, "Using the Borland Database Engine."
- The ADO page of the Component palette contains components that use ActiveX Data Objects (ADO) to access database information through OLEDB. ADO is a Microsoft Standard. There is a broad range of ADO drivers available for connecting to different database servers. Using ADO-based components lets you

integrate your application into an ADO-based environment (for example, making use of ADO-based application servers). For more information about using the ADO components, see Chapter 27, “Working with ADO components.”

- The dbExpress page of the Component palette contains components that use dbExpress to access database information. dbExpress is a lightweight set of drivers that provide the fastest access to database information. In addition, dbExpress components support cross-platform development because they are also available on Linux. However, dbExpress database components also support the narrowest range of data manipulation functions. For more information about using the dbExpress components, see Chapter 28, “Using unidirectional datasets.”
- The InterBase page of the Component palette contains components that access InterBase databases directly, without going through a separate engine layer.
- The Data Access page of the Component palette contains components that can be used with any data access mechanism. This page includes *TClientDataset*, which can work with data stored on disk or, using the *TDataSetProvider* component also on this page, with components from one of the other groups. For more information about using client datasets, see Chapter 29, “Using client datasets.” For more information about *TDataSetProvider*, see Chapter 30, “Using provider components.”

**Note** Different versions of Delphi include different drivers for accessing database servers using the BDE, ADO, or dbExpress.

When designing a database application, you must decide which set of components to use. Each data access mechanism differs in its range of functional support, the ease of deployment, and the availability of drivers to support different database servers.

In addition to choosing a data access mechanism, you must choose a database server. There are different types of databases and you will want to consider the advantages and disadvantages of each type before settling on a particular database server.

All types of databases contain tables which store information. In addition, most (but not all) servers support additional features such as

- Database security
- Transactions
- Referential integrity, stored procedures, and triggers

## Types of databases

Relational database servers vary in the way they store information and in the way they allow multiple users to access that information simultaneously. Delphi provides support for two types of relational database server:

- **Remote database servers** reside on a separate machine. Sometimes, the data from a remote database server does not even reside on a single machine, but is distributed over several servers. Although remote database servers vary in the way they store information, they provide a common logical interface to clients. This common interface is Structured Query Language (SQL). Because you access them using SQL, they are sometimes called SQL servers. (Another name is Remote Database Management system, or RDBMS.) In addition to the common commands that make up SQL, most remote database servers support a unique “dialect” of SQL. Examples of SQL servers include InterBase, Oracle, Sybase, Informix, Microsoft SQL server, and DB2.

- **Local databases** reside on your local drive or on a local area network. They often have proprietary APIs for accessing the data. When they are shared by several users, they use file-based locking mechanisms. Because of this, they are sometimes called file-based databases. Examples of local databases include Paradox, dBASE, FoxPro, and Access.

Applications that use local databases are called **single-tiered applications** because the application and the database share a single file system. Applications that use remote database servers are called **two-tiered applications** or **multi-tiered applications** because the application and the database operate on independent systems (or tiers).

Choosing the type of database to use depends on several factors. For example, your data may already be stored in an existing database. If you are creating the database tables your application uses, you may want to consider the following questions:

- How many users will be sharing these tables? Remote database servers are designed for access by several users at the same time. They provide support for multiple users through a mechanism called transactions. Some local databases (such as Local InterBase) also provide transaction support, but many only provide file-based locking mechanisms, and some (such as client dataset files) provide no multi-user support at all.
- How much data will the tables hold? Remote database servers can hold more data than local databases. Some remote database servers are designed for warehousing large quantities of data while others are optimized for other criteria (such as fast updates).
- What type of performance (speed) do you require from the database? Local databases are usually faster than remote database servers because they reside on the same system as the database application. Different remote database servers are optimized to support different types of operations, so you may want to consider performance when choosing a remote database server.
- What type of support will be available for database administration? Local databases require less support than remote database servers. Typically, they are less expensive to operate because they do not require separately installed servers or expensive site licenses.

## Database security

Databases often contain sensitive information. Different databases provide security schemes for protecting that information. Some databases, such as Paradox and dBASE, only provide security at the table or field level. When users try to access protected tables, they are required to provide a password. Once users have been authenticated, they can see only those fields (columns) for which they have permission.

Most SQL servers require a password and user name to use the database server at all. Once the user has logged in to the database, that username and password determine which tables can be used. For information on providing passwords to SQL servers, see "Controlling server login" on page 23-4.

When designing database applications, you must consider what type of authentication is required by your database server. Often, applications are designed to hide the explicit database login from users, who need only log in to the application itself. If you do not want to require your users to provide a database password, you must either use a database that does not require one or you must provide the password and username to the server programmatically. When providing the password programmatically, care must be taken that security can't be breached by reading the password from the application.

If you require your user to supply a password, you must consider when the password is required. If you are using a local database but intend to scale up to a larger SQL server later, you may want to prompt for the password at the point when you will eventually log in to the SQL database, rather than when opening individual tables.

If your application requires multiple passwords because you must log in to several protected systems or databases, you can have your users provide a single master password that is used to access a table of passwords required by the protected systems. The application then supplies passwords programmatically, without requiring the user to provide multiple passwords.

In multi-tiered applications, you may want to use a different security model altogether. You can use HTTPs, CORBA, or COM+ to control access to middle tiers, and let the middle tiers handle all details of logging into database servers.

## Transactions

A transaction is a group of actions that must all be carried out successfully on one or more tables in a database before they are committed (made permanent). If any of the actions in the group fails, then all actions are rolled back (undone).

Transactions ensure that

- All updates in a single transaction are either committed or aborted and rolled back to their previous state. This is referred to as **atomicity**.
- A transaction is a correct transformation of the system state, preserving the state invariants. This is referred to as **consistency**.

- Concurrent transactions do not see each other's partial or uncommitted results, which might create inconsistencies in the application state. This is referred to as **isolation**.
- Committed updates to records survive failures, including communication failures, process failures, and server system failures. This is referred to as **durability**.

Thus, transactions protect against hardware failures that occur in the middle of a database command or set of commands. Transactional logging allows you to recover the durable state after disk media failures. Transactions also form the basis of multi-user concurrency control on SQL servers. When each user interacts with the database only through transactions, one user's commands can't disrupt the unity of another user's transaction. Instead, the SQL server schedules incoming transactions, which either succeed as a whole or fail as a whole.

Transaction support is not part of most local databases, although it is provided by local InterBase. In addition, the BDE drivers provide limited transaction support for some local databases. Database transaction support is provided by the component that represents the database connection. For details on managing transactions using a database connection component, see "Managing transactions" on page 23-6.

In multi-tiered applications, you can create transactions that include actions other than database operations or that span multiple databases. For details on using transactions in multi-tiered applications, see "Managing transactions in multi-tiered applications" on page 31-17.

## Referential integrity, stored procedures, and triggers

All relational databases have certain features in common that allow applications to store and manipulate data. In addition, databases often provide other, database-specific, features that can prove useful for ensuring consistent relationships between the tables in a database. These include

- **Referential integrity.** Referential integrity provides a mechanism to prevent master/detail relationships between tables from being broken. When the user attempts to delete a field in a master table which would result in orphaned detail records, referential integrity rules prevent the deletion or automatically delete the orphaned detail records.
- **Stored procedures.** Stored procedures are sets of SQL statements that are named and stored on an SQL server. Stored procedures usually perform common database-related tasks on the server, and sometimes return sets of records (datasets).
- **Triggers.** Triggers are sets of SQL statements that are automatically executed in response to a particular command.

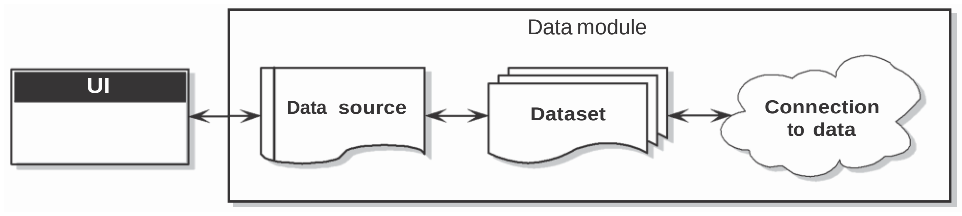
# Database architecture

Database applications are built from user interface elements, components that represent database information (datasets), and components that connect these to each other and to the source of the database information. How you organize these pieces is the architecture of your database application.

## General structure

While there are many distinct ways to organize the components in a database application, most follow the general scheme illustrated in Figure 19.1:

**Figure 19.1** Generic Database Architecture



### The user interface form

It is a good idea to isolate the user interface on a form that is completely separate from the rest of the application. This has several advantages. By isolating the user interface from the components that represent the database information itself, you introduce a greater flexibility into your design: Changes to the way you manage the database information do not require you to rewrite your user interface, and changes to the user interface do not require you to change the portion of your application that works with the database. In addition, this type of isolation lets you develop common forms that you can share between multiple applications, thereby providing a consistent user interface. By storing links to well-designed forms in the Object Repository, you and other developers can build on existing foundations rather than starting over from scratch for each new project. Sharing forms also makes it possible for you to develop corporate standards for application interfaces. For more information about creating the user interface for a database application, see "Designing the user interface" on page 19-15.

### The data module

If you have isolated your user interface into its own form, you can use a data module to house the components that represent database information (datasets), and the components that connect these datasets to the other parts of your application. Like the user interface forms, you can share data modules in the Object Repository so that they can be reused or shared between applications.

### The data source

The first item in the data module is a data source. The data source acts as a conduit between the user interface and a dataset that represents information from a database. Several data-aware controls on a form can share a single data source, in which case the display in each control is synchronized so that as the user scrolls through records, the corresponding value in the fields for the current record is displayed in each control.

### The dataset

The heart of your database application is the dataset. This component represents a set of records from the underlying database. These records can be the data from a single database table, a subset of the fields or records in a table, or information from more than one table joined into a single view. By using datasets, your application logic is buffered from restructuring of the physical tables in your databases. When the underlying database changes, you might need to alter the way the dataset component specifies the data it contains, but the rest of your application can continue to work without alteration. For more information on the common properties and methods of datasets, see Chapter 24, "Understanding datasets."

### The data connection

Different types of datasets use different mechanisms for connecting to the underlying database information. These different mechanisms, in turn, make up the major differences in the architecture of the database applications you can build. There are four basic mechanisms for connecting to the data:

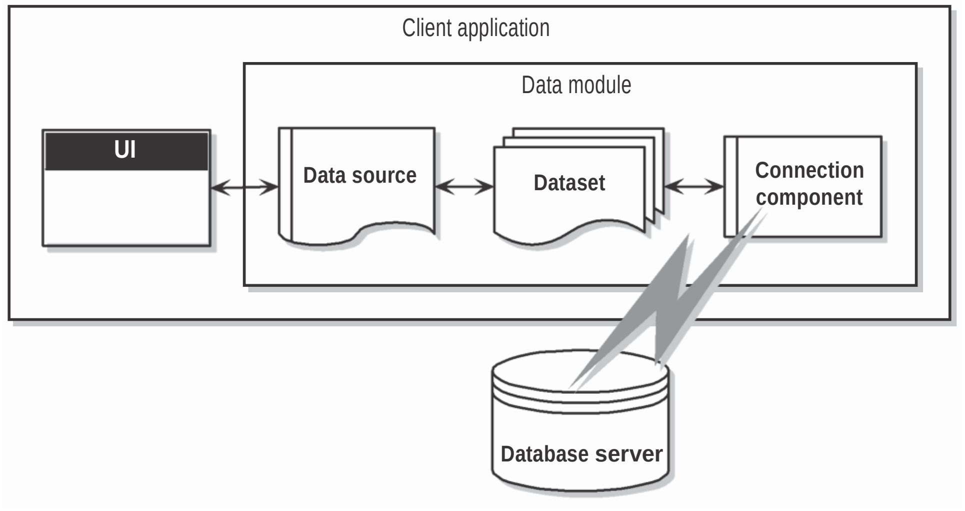
- Connecting directly to a database server. Most datasets use a descendant of *TCustomConnection* to represent the connection to a database server.
- Using a dedicated file on disk. Client datasets support the ability to work with a dedicated file on disk. No separate connection component is needed when working with a dedicated file because the client dataset itself knows how to read from and write to the file.
- Connecting to another dataset. Client datasets can work with data provided by another dataset. A *TDataSetProvider* component serves as an intermediary between the client dataset and its source dataset. This dataset provider can reside in the same data module as the client dataset, or it can be part of an application server running on another machine. If the provider is part of an application server, you also need a special descendant of *TCustomConnection* to represent the connection to the application server.
- Obtaining data from an RDS DataSpace object. ADO datasets can use a *TRDSConnection* component to marshal data in multi-tier database applications that are built using ADO-based application servers.

Sometimes, these mechanisms can be combined in a single application.

## Connecting directly to a database server

The most common database architecture is the one where the dataset uses a connection component to establish a connection to a database server. The dataset then fetches data directly from the server and posts edits directly to the server. This is illustrated in Figure 19.2.

**Figure 19.2** Connecting directly to the database server



Each type of dataset uses its own type of connection component, which represents a single data access mechanism:

- If the dataset is a BDE dataset such as *TTable*, *TQuery*, or *TStoredProc*, the connection component is a *TDataBase* object. You connect the dataset to the database component by setting its *Database* property. You do not need to explicitly add a database component when using BDE dataset. If you set the dataset's *DatabaseName* property, a database component is created for you automatically at runtime.
- If the dataset is an ADO dataset such as *TADODataSet*, *TADOTable*, *TADOQuery*, or *TADOStoredProc*, the connection component is a *TADOConnection* object. You connect the dataset to the ADO connection component by setting its *ADOConnection* property. As with BDE datasets, you do not need to explicitly add the connection component: instead you can set the dataset's *ConnectionString* property.

- If the dataset is a dbExpress dataset such as *TSQLDataSet*, *TSQLTable*, *TSQLQuery*, or *TSQLStoredProc*, the connection component is a *TSQLConnection* object. You connect the dataset to the SQL connection component by setting its *SQLConnection* property. When using dbExpress datasets, you must explicitly add the connection component. Another difference between dbExpress datasets and the other datasets is that dbExpress datasets are always read-only and unidirectional: This means you can only navigate by iterating through the records in order, and you can't use the dataset methods that support editing.
- If the dataset is an InterBase Express dataset such as *TIBDataSet*, *TIBTable*, *TIBQuery*, or *TIBStoredProc*, the connection component is a *TIBDatabase* object. You connect the dataset to the IB database component by setting its *Database* property. As with dbExpress datasets, you must explicitly add the connection component.

In addition to the components listed above, you can use a specialized client dataset such as *TBDEClientDataSet, TSimpleDataSet*, or *TIBClientDataSet* with a database connection component. When using one of these client datasets, specify the appropriate type of connection component as the value of the *DBConnection* property.

Although each type of dataset uses a different connection component, they all perform many of the same tasks and surface many of the same properties, methods, and events. For more information on the commonalities among the various database connection components, see Chapter 23, "Connecting to databases."

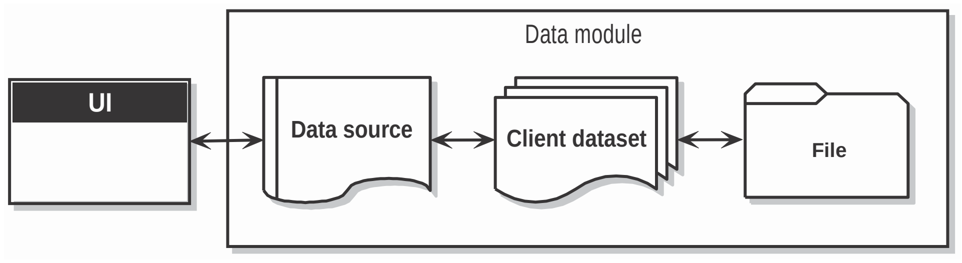
This architecture represents either a single-tiered or two-tiered application, depending on whether the database server is a local database such or a remote database server. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

**Note** The connection components or drivers needed to create two-tiered applications are not available in all version of Delphi.

## Using a dedicated file on disk

The simplest form of database application you can write does not use a database server at all. Instead, it uses MyBase, the ability of client datasets to save themselves to a file and to load the data from a file. This architecture is illustrated in Figure 19.3:

**Figure 19.3** A file-based database application



When using this file-based approach, your application writes changes to disk using the client dataset's *SaveToFile* method. *SaveToFile* takes one parameter, the name of the file which is created (or overwritten) containing the table. When you want to read a table previously written using the *SaveToFile* method, use the *LoadFromFile* method. *LoadFromFile* also takes one parameter, the name of the file containing the table.

If you always load to and save from the same file, you can use the *FileName* property instead of the *SaveToFile* and *LoadFromFile* methods. When *FileName* is set to a valid file name, the data is automatically loaded from the file when the client dataset is opened and saved to the file when the client dataset is closed.

This simple file-based architecture is a single-tiered application. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

The file-based approach has the benefit of simplicity. There is no database server to install, configure, or deploy (If you do not statically link in midaslib.dcu, the client dataset does require midas.dll). There is no need for site licenses or database administration.

In addition, some versions of Delphi let you convert between arbitrary XML documents and the data packets that are used by a client dataset. Thus, the file-based approach can be used to work with XML documents as well as dedicated datasets. For information about converting between XML documents and client dataset data packets, see Chapter 32, "Using XML in database applications."

The file-based approach offers no support for multiple users. The dataset should be dedicated entirely to the application. Data is saved to files on disk, and loaded at a later time, but there is no built-in protection to prevent multiple users from overwriting each other's data files.

For more information about using a client dataset with data stored on disk, see "Using a client dataset with file-based data" on page 29-33.

## Connecting to another dataset

There are specialized client datasets that use the BDE or *dbExpress* to connect to a database server. These specialized client datasets are, in fact, composite components that include another dataset internally to access the data and an internal provider component to package the data from the source dataset and to apply updates back to the database server. These composite components require some additional overhead, but provide certain benefits:

- Client datasets provide the most robust way to work with cached updates. By default, other types of datasets post edits directly to the database server. You can reduce network traffic by using a dataset that caches updates locally and applies them all later in a single transaction. For information on the advantages of using client datasets to cache updates, see "Using a client dataset to cache updates" on page 29-16.

- Client datasets can apply edits directly to a database server when the dataset is read-only. When using *dbExpress*, this is the only way to edit the data in the dataset (it is also the only way to navigate freely in the data when using *dbExpress*). Even when not using *dbExpress*, the results of some queries and all stored procedures are read-only. Using a client dataset provides a standard way to make such data editable.
- Because client datasets can work directly with dedicated files on disk, using a client dataset can be combined with a file-based model to allow for a flexible "briefcase" application. For information on the briefcase model, see "Combining approaches" on page 19-14.

In addition to these specialized client datasets, there is a generic client dataset (*TClientDataSet*), which does not include an internal dataset and dataset provider. Although *TClientDataSet* has no built-in database access mechanism, you can connect it to another, external, dataset from which it fetches data and to which it sends updates. Although this approach is a bit more complicated, there are times when it is preferable:

- Because the source dataset and dataset provider are external, you have more control over how they fetch data and apply updates. For example, the provider component surfaces a number of events that are not available when using a specialized client dataset to access data.
- When the source dataset is external, you can link it in a master/detail relationship with another dataset. An external provider automatically converts this arrangement into a single dataset with nested details. When the source dataset is internal, you can't create nested detail sets this way.
- Connecting a client dataset to an external dataset is an architecture that easily scales up to multiple tiers. Because the development process can get more involved and expensive as the number of tiers increases, you may want to start developing your application as a single-tiered or two-tiered application. As the amount of data, the number of users, and the number of different applications accessing the data grows, you may later need to scale up to a multi-tiered architecture. If you think you may eventually use a multi-tiered architecture, it can be worthwhile to start by using a client dataset with an external source dataset. This way, when you move the data access and manipulation logic to a middle tier, you protect your development investment because the code can be reused as your application grows.
- *TClientDataSet* can link to any source dataset. This means you can use custom datasets (third-party components) for which there is no corresponding specialized client dataset. Some versions of Delphi even include special provider components that connect a client dataset to an XML document rather than another dataset. (This works the same way as connecting a client dataset to another (source) dataset, except that the XML provider uses an XML document rather than a dataset. For information about these XML providers, see "Using an XML document as the source for a provider" on page 32-8.)

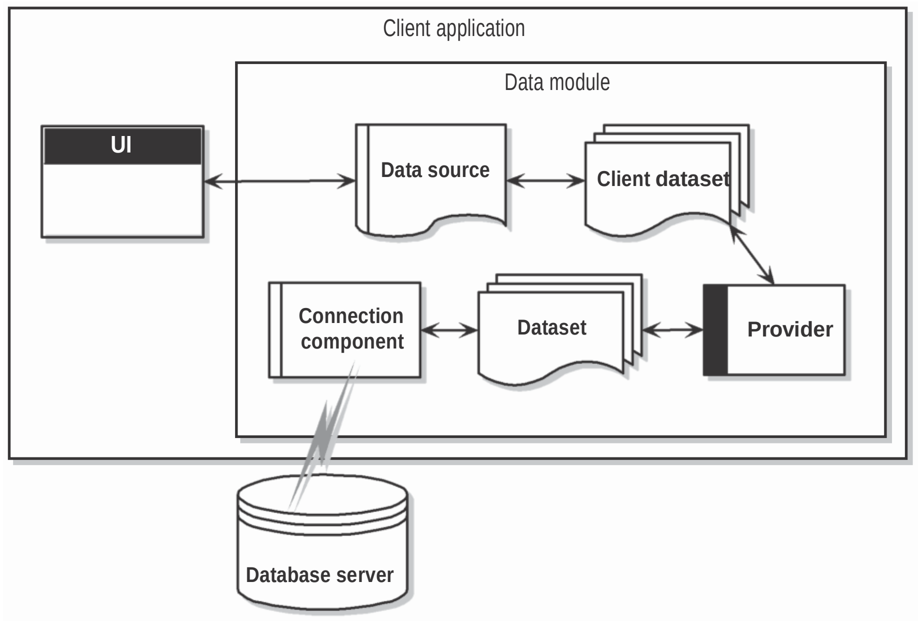
There are two versions of the architecture that connects a client dataset to an external dataset:

- Connecting a client dataset to another dataset in the same application.
- Using a multi-tiered architecture.

## Connecting a client dataset to another dataset in the same application

By using a provider component, you can connect *TClientDataSet* to another (source) dataset. The provider packages database information into transportable data packets (which can be used by client datasets) and applies updates received in delta packets (which client datasets create) back to a database server. The architecture for this is illustrated in Figure 19.4.

**Figure 19.4** Architecture combining a client dataset and another dataset



This architecture represents either a single-tiered or two-tiered application, depending on whether the database server is a local database or a remote database server. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

To link the client dataset to the provider, set its *ProviderName* property to the name of the provider component. The provider must be in the same data module as the client dataset. To link the provider to the source dataset, set its *DataSet* property.

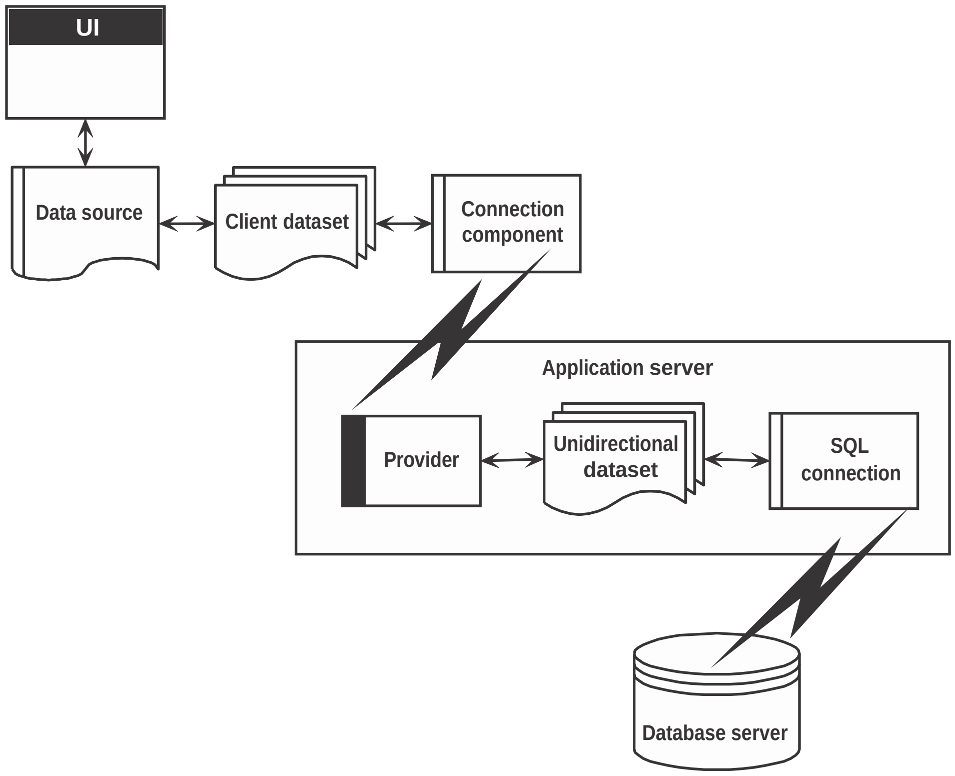
Once the client dataset is linked to the provider and the provider is linked to the source dataset, these components automatically handle all the details necessary for fetching, displaying, and navigating through the database records (assuming the source dataset is connected to a database). To apply user edits back to the database, you need only call the client dataset's *ApplyUpdates* method.

For more information on using a client dataset with a provider, see "Using a client dataset with a provider" on page 29-24.

## Using a multi-tiered architecture

When the database information includes complicated relationships between several tables, or when the number of clients grows, you may want to use a multi-tiered application. Multi-tiered applications have middle tiers between the client application and database server. The architecture for this is illustrated in Figure 19.5.

**Figure 19.5** Multi-tiered database architecture



The preceding figure represents three-tiered application. The logic that manipulates database information is on a separate system, or tier. This middle tier centralizes the logic that governs your database interactions so there is centralized control over data relationships. This allows different client applications to use the same data, while ensuring consistent data logic. It also allows for smaller client applications because much of the processing is off-loaded onto the middle tier. These smaller client applications are easier to install, configure, and maintain. Multi-tiered applications can also improve performance by spreading data-processing over several systems.

The multi-tiered architecture is very similar to the previous model. It differs mainly in that source dataset that connects to the database server and the provider that acts as an intermediary between that source dataset and the client dataset have both moved to a separate application. That separate application is called the application server (or sometimes the “remote data broker”).

Because the provider has moved to a separate application, the client dataset can no longer connect to the source dataset by simply setting its *ProviderName* property. In addition, it must use some type of connection component to locate and connect to the application server.

There are several types of connection components that can connect a client dataset to an application server. They are all descendants of *TCustomRemoteServer*, and differ primarily in the communication protocol they use (TCP/IP, HTTP, DCOM, SOAP, or CORBA). Link the client dataset to its connection component by setting the *RemoteServer* property.

The connection component establishes a connection to the application server and returns an interface that the client dataset uses to call the provider specified by its *ProviderName* property. Each time the client dataset calls the application server, it passes the value of *ProviderName*, and the application server forwards the call to the provider.

For more information about connecting a client dataset to an application server, see Chapter 31, “Creating multi-tiered applications.”

## Combining approaches

The previous sections describe several architectures you can use when writing database applications. There is no reason, however, why you can’t combine two or more of the available architectures in a single application. In fact, some combinations can be extremely powerful.

For example, you can combine the disk-based architecture described in “Using a dedicated file on disk” on page 19-9 with another approach such as those described in “Connecting a client dataset to another dataset in the same application” on page 19-12 or “Using a multi-tiered architecture” on page 19-13. These combinations are easy because all models use a client dataset to represent the data that appears in the user interface. The result is called the briefcase model (or sometimes the disconnected model, or mobile computing).

The briefcase model is useful in a situation such as the following: An onsite company database contains customer contact data that sales representatives can use and update in the field. While onsite, sales representatives download information from the database. Later, they work with it on their laptops as they fly across the country, and even update records at existing or new customer sites. When the sales representatives return onsite, they upload their data changes to the company database for everyone to use.

When operating on site, the client dataset in a briefcase model application fetches its data from a provider. The client dataset is therefore connected to the database server and can, through the provider, fetch server data and send updates back to the server. Before disconnecting from the provider, the client dataset saves its snapshot of the information to a file on disk. While offsite, the client dataset loads its data from the file, and saves any changes back to that file. Finally, back onsite, the client dataset reconnects to the provider so that it can apply its updates to the database server or refresh its snapshot of the data.

## Designing the user interface

The Data Controls page of the Component palette provides a set of data-aware controls that represent data from fields in a database record, and can permit users to edit that data and post changes back to the database. Using data-aware controls, you can build your database application's user interface (UI) so that information is visible and accessible to users. For more information on data-aware controls see Chapter 20, "Using data controls."

In addition to the basic data controls, you may also want to introduce other elements into your user interface:

- You may want your application to analyze the data contained in a database. Applications that analyze data do more than just display the data in a database, they also summarize the information in useful formats to help users grasp the impact of that data.
- You may want to print reports that provide a hard copy of the information displayed in your user interface.
- You may want to create a user interface that can be viewed from Web browsers. The simplest Web-based database applications are described in "Using database information in responses" on page 34-18. In addition, you can combine the Web-based approach with the multi-tiered architecture, as described in "Writing Web-based client applications."

### Analyzing data

Some database applications do not present database information directly to the user. Instead, they analyze and summarize information from databases so that users can draw conclusions from the data.

The *TDBChart* component on the Data Controls page of the Component palette lets you present database information in a graphical format that enables users to quickly grasp the import of database information.

In addition, some versions of Delphi include a Decision Cube page on the Component palette. It contains six components that let you perform data analysis and cross-tabulations on data when building decision support applications. For more information about using the Decision Cube components, see Chapter 22, "Using decision support components."

If you want to build your own components that display data summaries based on various grouping criteria, you can use maintained aggregates with a client dataset. For more information about using maintained aggregates, see "Using maintained aggregates" on page 29-11.

## Writing reports

If you want to let your users print database information from the datasets in your application, you can use Rave Reports, as described in Chapter 21, "Creating reports with Rave Reports."

Chapter

20

# Using data controls

The Data Controls page of the Component palette provides a set of data-aware controls that represent data from fields in a database record, and, if the dataset allows it, enable users to edit that data and post changes back to the database. By placing data controls onto the forms in your database application, you can build your database application's user interface (UI) so that information is visible and accessible to users.

The data-aware controls you add to your user interface depend on several factors, including the following:

- The type of data you are displaying. You can choose between controls that are designed to display and edit plain text, controls that work with formatted text, controls for graphics, multimedia elements, and so on. Controls that display different types of information are described in "Displaying a single record" on page 20-7.
- How you want to organize the information. You may choose to display information from a single record on the screen, or list the information from multiple records using a grid. "Choosing how to organize the data" on page 20-7 describes some of the possibilities.
- The type of dataset that supplies data to the controls. You want to use controls that reflect the limitations of the underlying dataset. For example, you would not use a grid with a unidirectional dataset because unidirectional datasets can only supply a single record at a time.
- How (or if) you want to let users navigate through the records of datasets and add or edit data. You may want to add your own controls or mechanisms to navigate and edit, or you may want to use a built-in control such as a data navigator. For more information about using a data navigator, see "Navigating and manipulating records" on page 20-29.

**Note** More complex data-aware controls for decision support are discussed in Chapter 22, "Using decision support components."

Regardless of the data-aware controls you choose to add to your interface, certain common features apply. These are described below.

## Using common data control features

The following tasks are common to most data controls:

- Associating a data control with a dataset
- Editing and updating data
- Disabling and enabling data display
- Refreshing data display
- Enabling mouse, keyboard, and timer events

Data controls let you display and edit fields of data associated with the current record in a dataset. Table 20.1 summarizes the data controls that appear on the Data Controls page of the Component palette.

**Table 20.1** Data controls

| Data control | Description |
|---|---|
| *TDBGrid* | Displays information from a data source in a tabular format. Columns in the grid correspond to columns in the underlying table or query's dataset. Rows in the grid correspond to records. |
| *TDBNavigator* | Navigates through data records in a dataset. updating records, posting records, deleting records, canceling edits to records, and refreshing data display. |
| *TDBText* | Displays data from a field as a label. |
| *TDBEdit* | Displays data from a field in an edit box. |
| *TDBMemo* | Displays data from a memo or BLOB field in a scrollable, multi-line edit box. |
| *TDBImage* | Displays graphics from a data field in a graphics box. |
| *TDBListBox* | Displays a list of items from which to update a field in the current data record. |
| *TDBComboBox* | Displays a list of items from which to update a field, and also permits direct text entry like a standard data-aware edit box. |
| *TDBCheckBox* | Displays a check box that indicates the value of a Boolean field. |
| *TDBRadioGroup* | Displays a set of mutually exclusive options for a field. |
| *TDBLookupListBox* | Displays a list of items looked up from another dataset based on the value of a field. |
| *TDBLookupComboBox* | Displays a list of items looked up from another dataset based on the value of a field, and also permits direct text entry like a standard data-aware edit box. |
| *TDBCtrlGrid* | Displays a configurable, repeating set of data-aware controls within a grid. |
| *TDBRichEdit* | Displays formatted data from a field in an edit box. |

Data controls are data-aware at design time. When you associate the data control with an active dataset while building an application, you can immediately see live data in the control. You can use the Fields editor to scroll through a dataset at design time to verify that your application displays data correctly without having to compile and run the application. For more information about the Fields editor, see "Creating persistent fields" on page 25-4.

At runtime, data controls display data and, if your application, the control, and the dataset all permit it, a user can edit data through the control.

## Associating a data control with a dataset

Data controls connect to datasets by using a data source. A data source component (*TDataSource*) acts as a conduit between the control and a dataset containing data. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component in order for their data to be displayed and manipulated in data-aware controls on a form.

**Note** Data source components are also required for linking unnested datasets in master-detail relationships.

To associate a data control with a dataset,

1. Place a dataset in a data module (or on a form), and set its properties as appropriate.
2. Place a data source in the same data module (or form). Using the Object Inspector, set its *DataSet* property to the dataset you placed in step 1.
3. Place a data control from the Data Access page of the Component palette onto a form.
4. Using the Object Inspector, set the *DataSource* property of the control to the data source component you placed in step 2.
5. Set the *DataField* property of the control to the name of a field to display, or select a field name from the drop-down list for the property. This step does not apply to *TDBGrid*, *TDBCtrlGrid*, and *TDBNavigator* because they access all available fields in the dataset.
6. Set the *Active* property of the dataset to *True* to display data in the control.

### Changing the associated dataset at runtime

In the preceding example, the datasource was associated with its dataset by setting the *DataSet* property at design time. At runtime, you can switch the dataset for a data source component as needed. For example, the following code swaps the dataset for the *CustSource* data source component between the dataset components named *Customers* and *Orders*:

```
with CustSource do begin
  if (DataSet = Customers) then
    DataSet := Orders
  else
    DataSet := Customers;
end;
```

You can also set the *DataSet* property to a dataset on another form to synchronize the data controls on two forms. For example:

```
procedure TForm2.FormCreate (Sender : TObject);
begin
  DataSource1.Dataset := Form1.Table1;
end;
```

### Enabling and disabling the data source

The data source has an *Enabled* property that determines if it is connected to its dataset. When *Enabled* is *True*, the data source is connected to a dataset.

You can temporarily disconnect a single data source from its dataset by setting *Enabled* to *False*. When *Enabled* is *False*, all data controls attached to the data source component go blank and become inactive until *Enabled* is set to *True*. It is recommended, however, to control access to a dataset through a dataset component's *DisableControls* and *EnableControls* methods because they affect all attached data sources.

### Responding to changes mediated by the data source

Because the data source provides the link between the data control and its dataset, it mediates all of the communication that occurs between the two. Typically, the data-aware control automatically responds to changes in the dataset. However, if your user interface is using controls that are not data-aware, you can use the events of a data source component to manually provide the same sort of response.

The *OnDataChange* event occurs whenever the data in a record may have changed, including field edits or when the cursor moves to a new record. This event is useful for making sure the control reflects the current field values in the dataset, because it is triggered by all changes. Typically, an *OnDataChange* event handler refreshes the value of a non-data-aware control that displays field data.

The *OnUpdateData* event occurs when the data in the current record is about to be posted. For instance, an *OnUpdateData* event occurs after *Post* is called, but before the data is actually posted to the underlying database server or local cache.

The *OnStateChange* event occurs when the state of the dataset changes. When this event occurs, you can examine the dataset's *State* property to determine its current state.

For example, the following *OnStateChange* event handler enables or disables buttons or menu items based on the current state:

```
procedure Form1.DataSource1.StateChange(Sender: TObject);
begin
  CustTableEditBtn.Enabled := (CustTable.State = dsBrowse);
  CustTableCancelBtn.Enabled := CustTable.State in [dsInsert, dsEdit, dsSetKey];
  CustTableActivateBtn.Enabled := CustTable.State in [dsInactive];
  ƒ
end;
```

**Note** For more information about dataset states, see "Determining dataset states" on page 24-3.

## Editing and updating data

All data controls except the navigator display data from a database field. In addition, you can use them to edit and update data as long as the underlying dataset allows it.

**Note** Unidirectional datasets never permit users to edit and update data.

### Enabling editing in controls on user entry

A dataset must be in *dsEdit* state to permit editing to its data. If the data source's *AutoEdit* property is *True* (the default), the data control handles the task of putting the dataset into *dsEdit* mode as soon as the user tries to edit its data.

If *AutoEdit* is *False*, you must provide an alternate mechanism for putting the dataset into edit mode. One such mechanism is to use a *TDBNavigator* control with an *Edit* button, which lets users explicitly put the dataset into edit mode. For more information about *TDBNavigator*, see "Navigating and manipulating records" on page 20-29. Alternately, you can write code that calls the dataset's *Edit* method when you want to put the dataset into edit mode.

### Editing data in a control

A data control can only post edits to its associated dataset if the dataset's *CanModify* property is *True*. *CanModify* is always *False* for unidirectional datasets. Some datasets have a *ReadOnly* property that lets you specify whether *CanModify* is *True*.

**Note** Whether a dataset can update data depends on whether the underlying database table permits updates.

Even if the dataset's *CanModify* property is *True*, the *Enabled* property of the data source that connects the dataset to the control must be *True* as well before the control can post updates back to the database table. The *Enabled* property of the data source determines whether the control can display field values from the dataset, and therefore also whether a user can edit and post values. If *Enabled* is ***True*** (the default), controls can display field values.

Finally, you can control whether the user can even enter edits to the data that is displayed in the control. The *ReadOnly* property of the data control determines if a user can edit the data displayed by the control. If *False* (the default), users can edit data. Clearly, you will want to ensure that the control's *ReadOnly* property is *True* when the dataset's *CanModify* property is *False*. Otherwise, you give users the false impression that they can affect the data in the underlying database table.

In all data controls except *TDBGrid*, when you modify a field, the modification is copied to the underlying dataset when you Tab from the control. If you press Esc before you Tab from a field, the data control abandons the modifications, and the value of the field reverts to the value it held before any modifications were made.

In *TDBGrid*, modifications are posted when you move to a different record; you can press Esc in any record of a field before moving to another record to cancel all changes to the record.

When a record is posted, Delphi checks all data-aware controls associated with the dataset for a change in status. If there is a problem updating any fields that contain modified data, Delphi raises an exception, and no modifications are made to the record.

**Note** If your application caches updates (for example, using a client dataset), all modifications are posted to an internal cache. These modifications are not applied to the underlying database table until you call the dataset's *ApplyUpdates* method.

## Disabling and enabling data display

When your application iterates through a dataset or performs a search, you should temporarily prevent refreshing of the values displayed in data-aware controls each time the current record changes. Preventing refreshing of values speeds the iteration or search and prevents annoying screen-flicker.

*DisableControls* is a dataset method that disables display for all data-aware controls linked to a dataset. As soon as the iteration or search is over, your application should immediately call the dataset's *EnableControls* method to re-enable display for the controls.

Usually you disable controls before entering an iterative process. The iterative process itself should take place inside a **try...finally** statement so that you can re-enable controls even if an exception occurs during processing. The **finally** clause should call *EnableControls*. The following code illustrates how you might use *DisableControls* and *EnableControls* in this manner:

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets EOF False }
  while not CustTable.EOF do { Cycle until EOF is True }
  begin
    { Process each record here }
    f
    CustTable.Next; { EOF False on success; EOF True when Next fails on last record }
  end;
```

```
finally
  CustTable.EnableControls;
end;
```

### Refreshing data display

The *Refresh* method for a dataset flushes local buffers and re-fetches data for an open dataset. You can use this method to update the display in data-aware controls if you think that the underlying data has changed because other applications have simultaneous access to the data used in your application. If you are using cached updates, before you refresh the dataset you must apply any updates the dataset has currently cached.

Refreshing can sometimes lead to unexpected results. For example, if a user is viewing a record deleted by another application, then the record disappears the moment your application calls *Refresh*. Data can also appear to change if another user changes a record after you originally fetched the data and before you call *Refresh*.

### Enabling mouse, keyboard, and timer events

The *Enabled* property of a data control determines whether it responds to mouse, keyboard, or timer events, and passes information to its data source. The default setting for this property is *True*.

To prevent mouse, keyboard, or timer events from reaching a data control, set its *Enabled* property to *False*. When *Enabled* is *False*, the data source that connects the control to its dataset does not receive information from the data control. The data control continues to display data, but the text displayed in the control is dimmed.

## Choosing how to organize the data

When you build the user interface for your database application, you have choices to make about how you want to organize the display of information and the controls that manipulate that information.

One of the first decisions to make is whether you want to display a single record at a time, or multiple records.

In addition, you will want to add controls to navigate and manipulate records. The *TDBNavigator* control provides built-in support for many of the functions you may want to perform.

### Displaying a single record

In many applications, you may only want to provide information about a single record of data at a time. For example, an order-entry application may display the information about a single order without indicating what other orders are currently logged. This information probably comes from a single record in an orders dataset.

Applications that display a single record are usually easy to read and understand, because all database information is about the same thing (in the previous case, the same order). The data-aware controls in these user interfaces represent a single field from a database record. The Data Controls page of the Component palette provides a wide selection of controls to represent different kinds of fields. These controls are typically data-aware versions of other controls that are available on the Component palette. For example, the *TDBEdit* control is a data-aware version of the standard *TEdit* control which enables users to see and edit a text string.

Which control you use depends on the type of data (text, formatted text, graphics, boolean information, and so on) contained in the field.

### Displaying data as labels

*TDBText* is a read-only control similar to the *TLabel* component on the Standard page of the Component palette. A *TDBText* control is useful when you want to provide display-only data on a form that allows user input in other controls. For example, suppose a form is created around the fields in a customer list table, and that once the user enters a street address, city, and state or province information in the form, you use a dynamic lookup to automatically determine the zip code field from a separate table. A *TDBText* component tied to the zip code table could be used to display the zip code field that matches the address entered by the user.

*TDBText* gets the text it displays from a specified field in the current record of a dataset. Because *TDBText* gets its text from a dataset, the text it displays is dynamic—the text changes as the user navigates the database table. Therefore you cannot specify the display text of *TDBText* at design time as you can with *TLabel*.

**Note** When you place a *TDBText* component on a form, make sure its *AutoSize* property is *True* (the default) to ensure that the control resizes itself as necessary to display data of varying widths. If *AutoSize* is *False*, and the control is too small, data display is clipped.

### Displaying and editing fields in an edit box

*TDBEdit* is a data-aware version of an edit box component. *TDBEdit* displays the current value of a data field to which it is linked and permits it to be edited using standard edit box techniques.

For example, suppose *CustomersSource* is a *TDataSource* component that is active and linked to an open *TClientDataSet* called *CustomersTable*. You can then place a *TDBEdit* component on a form and set its properties as follows:

- *DataSource*: CustomersSource
- *DataField*: CustNo

The data-aware edit box component immediately displays the value of the current row of the *CustNo* column of the *CustomersTable* dataset, both at design time and at runtime.

### Displaying and editing text in a memo control

*TDBMemo* is a data-aware component—similar to the standard *TMemo* component—that can display lengthy text data. *TDBMemo* displays multi-line text, and permits a user to enter multi-line text as well. You can use *TDBMemo* controls to display large text fields or text data contained in binary large object (BLOB) fields.

By default, *TDBMemo* permits a user to edit memo text. To prevent editing, set the *ReadOnly* property of the memo control to *True*. To display tabs and permit users to enter them in a memo, set the *WantTabs* property to *True*. To limit the number of characters users can enter into the database memo, use the *MaxLength* property. The default value for *MaxLength* is 0, meaning that there is no character limit other than that imposed by the operating system.

Several properties affect how the database memo appears and how text is entered. You can supply scroll bars in the memo with the *ScrollBars* property. To prevent word wrap, set the *WordWrap* property to ***False***. The *Alignment* property determines how the text is aligned within the control. Possible choices are *taLeftJustify* (the default), *taCenter*, and *taRightJustify*. To change the font of the text, use the *Font* property.

At runtime, users can cut, copy, and paste text to and from a database memo control. You can accomplish the same task programmatically by using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods.

Because the *TDBMemo* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes to scroll through data records, *TDBMemo* has an *AutoDisplay* property that controls whether the accessed data should be displayed automatically. If you set *AutoDisplay* to ***False***, *TDBMemo* displays the field name rather than actual data. Double-click inside the control to view the actual data.

### Displaying and editing text in a rich edit memo control

*TDBRichEdit* is a data-aware component—similar to the standard *TRichEdit* component—that can display formatted text stored in a binary large object (BLOB) field. *TDBRichEdit* displays formatted, multi-line text, and permits a user to enter formatted multi-line text as well.

**Note** While *TDBRichEdit* provides properties and methods to enter and work with rich text, it does not provide any user interface components to make these formatting options available to the user. Your application must implement the user interface to surface rich text capabilities.

By default, *TDBRichEdit* permits a user to edit memo text. To prevent editing, set the *ReadOnly* property of the rich edit control to *True*. To display tabs and permit users to enter them in a memo, set the *WantTabs* property to *True*. To limit the number of characters users can enter into the database memo, use the *MaxLength* property. The default value for MaxLength is 0, meaning that there is no character limit other than that imposed by the operating system.

Because the *TDBRichEdit* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes to scroll through data records, *TDBRichEdit* has an *AutoDisplay* property that controls whether the accessed data should be displayed automatically. If you set *AutoDisplay* to *False*, *TDBRichEdit* displays the field name rather than actual data. Double-click inside the control to view the actual data.

### Displaying and editing graphics fields in an image control

*TDBImage* is a data-aware control that displays graphics contained in BLOB fields.

By default, *TDBImage* permits a user to edit a graphics image by cutting and pasting to and from the Clipboard using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods. You can, instead, supply your own editing methods attached to the event handlers for the control.

By default, an image control displays as much of a graphic as fits in the control, cropping the image if it is too big. You can set the *Stretch* property to *True* to resize the graphic to fit within an image control as it is resized.

Because the *TDBImage* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes scroll through data records, *TDBImage* has an *AutoDisplay* property that controls whether the accessed data should automatically displayed. If you set *AutoDisplay* to *False*, *TDBImage* displays the field name rather than actual data. Double-click inside the control to view the actual data.

### Displaying and editing data in list and combo boxes

There are four data controls that provide the user with a set of default data values to choose from at runtime. These are data-aware versions of standard list and combo box controls:

- *TDBListBox*, which displays a scrollable list of items from which a user can choose to enter in a data field. A data-aware list box displays a default value for a field in the current record and highlights its corresponding entry in the list. If the current row’s field value is not in the list, no value is highlighted in the list box. When a user selects a list item, the corresponding field value is changed in the underlying dataset.
- *TDBComboBox*, which combines the functionality of a data-aware edit control and a drop-down list. At runtime it can display a drop-down list from which a user can pick from a predefined set of values, and it can permit a user to enter an entirely different value.
- *TDBLookupListBox*, which behaves like *TDBListBox* except the list of display items is looked up in another dataset.
- *TDBLookupComboBox*, which behaves like *TDBComboBox* except the list of display items is looked up in another dataset.

**Note** At runtime, users can use an incremental search to find list box items. When the control has focus, for example, typing 'ROB' selects the first item in the list box beginning with the letters 'ROB'. Typing an additional 'E' selects the first item starting with 'ROBE', such as 'Robert Johnson'. The search is case-insensitive. Backspace and Esc cancel the current search string (but leave the selection intact), as does a two second pause between keystrokes.

### Using TDBListBox and TDBComboBox

When using *TDBListBox* or *TDBComboBox*, you must use the String List editor at design time to create the list of items to display. To bring up the String List editor, click the ellipsis button for the *Items* property in the Object Inspector. Then type in the items that you want to have appear in the list. At runtime, use the methods of the *Items* property to manipulate its string list.

When a *TDBListBox* or *TDBComboBox* control is linked to a field through its *DataField* property, the field value appears selected in the list. If the current value is not in the list, no item appears selected. However, *TDBComboBox* displays the current value for the field in its edit box, regardless of whether it appears in the *Items* list.

For *TDBListBox*, the *Height* property determines how many items are visible in the list box at one time. The *IntegralHeight* property controls how the last item can appear. If *IntegralHeight* is ***False*** (the default), the bottom of the list box is determined by the *ItemHeight* property, and the bottom item may not be completely displayed. If *IntegralHeight* is *True*, the visible bottom item in the list box is fully displayed.

For *TDBComboBox*, the *Style* property determines user interaction with the control. By default, *Style* is *csDropDown*, meaning a user can enter values from the keyboard, or choose an item from the drop-down list. The following properties determine how the *Items* list is displayed at runtime:

- *Style* determines the display style of the component:
  - *csDropDown* (default): Displays a drop-down list with an edit box in which the user can enter text. All items are strings and have the same height.
  - *csSimple*: Combines an edit control with a fixed size list of items that is always displayed. When setting *Style* to *csSimple*, be sure to increase the *Height* property so that the list is displayed.
  - *csDropDownList*: Displays a drop-down list and edit box, but the user cannot enter or change values that are not in the drop-down list at runtime.
  - *csOwnerDrawFixed* and *csOwnerDrawVariable*: Allows the items list to display values other than strings (for example, bitmaps) or to use different fonts for individual items in the list.
- *DropDownCount*: the maximum number of items displayed in the list. If the number of *Items* is greater than *DropDownCount*, the user can scroll the list. If the number of *Items* is less than *DropDownCount*, the list will be just large enough to display all the Items.

- *ItemHeight*: The height of each item when style is *csOwnerDrawFixed*.
- *Sorted*: If *True*, then the *Items* list is displayed in alphabetical order.

### Displaying and editing data in lookup list and combo boxes

Lookup list boxes and lookup combo boxes (*TDBLookupListBox* and *TDBLookupComboBox*) present the user with a restricted list of choices from which to set a valid field value. When a user selects a list item, the corresponding field value is changed in the underlying dataset.

For example, consider an order form whose fields are tied to the *OrdersTable*. *OrdersTable* contains a *CustNo* field corresponding to a customer ID, but *OrdersTable* does not have any other customer information. The *CustomersTable*, on the other hand, contains a *CustNo* field corresponding to a customer ID, and also contains additional information, such as the customer's company and mailing address. It would be convenient if the order form enabled a clerk to select a customer by company name instead of customer ID when creating an invoice. A *TDBLookupListBox* that displays all company names in *CustomersTable* enables a user to select the company name from the list, and set the *CustNo* on the order form appropriately.

These lookup controls derive the list of display items from one of two sources:

- **A lookup field defined for a dataset.**
  To specify list box items using a lookup field, the dataset to which you link the control must already define a lookup field. (This process is described in "Defining a lookup field" on page 25-9). To specify the lookup field for the list box items,

  **a** Set the *DataSource* property of the list box to the data source for the dataset containing the lookup field to use.

  **b** Choose the lookup field to use from the drop-down list for the *DataField* property.

  When you activate a table associated with a lookup control, the control recognizes that its data field is a lookup field, and displays the appropriate values from the lookup.

- **A secondary data source, data field, and key**.
  If you have not defined a lookup field for a dataset, you can establish a similar relationship using a secondary data source, a field value to search on in the secondary data source, and a field value to return as a list item. To specify a secondary data source for list box items,

  **a** Set the *DataSource* property of the list box to the data source for the control.

  **b** Choose a field into which to insert looked-up values from the drop-down list for the *DataField* property. The field you choose cannot be a lookup field.

  **c** Set the *ListSource* property of the list box to the data source for the dataset that contain the field whose values you want to look up.

**d** Choose a field to use as a lookup key from the drop-down list for the *KeyField* property. The drop-down list displays fields for the dataset associated with data source you specified in Step 3. The field you choose need not be part of an index, but if it is, lookup performance is even faster.

**e** Choose a field whose values to return from the drop-down list for the *ListField* property. The drop-down list displays fields for the dataset associated with the data source you specified in Step 3.

When you activate a table associated with a lookup control, the control recognizes that its list items are derived from a secondary source, and displays the appropriate values from that source.

To specify the number of items that appear at one time in a *TDBLookupListBox* control, use the *RowCount* property. The height of the list box is adjusted to fit this row count exactly.

To specify the number of items that appear in the drop-down list of *TDBLookupComboBox*, use the *DropDownRows* property instead.

**Note** You can also set up a column in a data grid to act as a lookup combo box. For information on how to do this, see "Defining a lookup list column" on page 20-21.

## Handling Boolean field values with check boxes

*TDBCheckBox* is a data-aware check box control. It can be used to set the values of Boolean fields in a dataset. For example, a customer invoice form might have a check box control that when checked indicates the customer is tax-exempt, and when unchecked indicates that the customer is not tax-exempt.

The data-aware check box control manages its checked or unchecked state by comparing the value of the current field to the contents of *ValueChecked* and *ValueUnchecked* properties. If the field value matches the *ValueChecked* property, the control is checked. Otherwise, if the field matches the *ValueUnchecked* property, the control is unchecked.

**Note** The values in *ValueChecked* and *ValueUnchecked* cannot be identical.

Set the *ValueChecked* property to a value the control should post to the database if the control is checked when the user moves to another record. By default, this value is set to "true," but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueChecked*. If any of the items matches the contents of that field in the current record, the check box is checked. For example, you can specify a *ValueChecked* string like:

```
DBCheckBox1.ValueChecked := 'True;Yes;On';
```

If the field for the current record contains values of "true," "Yes," or "On," then the check box is checked. Comparison of the field to *ValueChecked* strings is case-insensitive. If a user checks a box for which there are multiple *ValueChecked* strings, the first string is the value that is posted to the database.

Set the *ValueUnchecked* property to a value the control should post to the database if the control is not checked when the user moves to another record. By default, this value is set to "false," but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueUnchecked*. If any of the items matches the contents of that field in the current record, the check box is unchecked.

A data-aware check box is disabled whenever the field for the current record does not contain one of the values listed in the *ValueChecked* or *ValueUnchecked* properties.

If the field with which a check box is associated is a logical field, the check box is always checked if the contents of the field is *True*, and it is unchecked if the contents of the field is *False*. In this case, strings entered in the *ValueChecked* and *ValueUnchecked* properties have no effect on logical fields.

### Restricting field values with radio controls

*TDBRadioGroup* is a data-aware version of a radio group control. It enables you to set the value of a data field with a radio button control where there is a limited number of possible values for the field. The radio group includes one button for each value a field can accept. Users can set the value for a data field by selecting the desired radio button.

The *Items* property determines the radio buttons that appear in the group. *Items* is a string list. One radio button is displayed for each string in *Items*, and each string appears to the right of a radio button as the button's label.

If the current value of a field associated with a radio group matches one of the strings in the *Items* property, that radio button is selected. For example, if three strings, "Red," "Yellow," and "Blue," are listed for *Items*, and the field for the current record contains the value "Blue," then the third button in the group appears selected.

**Note** If the field does not match any strings in *Items*, a radio button may still be selected if the field matches a string in the *Values* property. If the field for the current record does not match any strings in *Items* or *Values*, no radio button is selected.

The *Values* property can contain an optional list of strings that can be returned to the dataset when a user selects a radio button and posts a record. Strings are associated with buttons in numeric sequence. The first string is associated with the first button, the second string with the second button, and so on. For example, suppose *Items* contains "Red," "Yellow," and "Blue," and *Values* contains "Magenta," "Yellow," and "Cyan." If a user selects the button labeled "Red," "Magenta" is posted to the database.

If strings for *Values* are not provided, the *Item* string for a selected radio button is returned to the database when a record is posted.

## Displaying multiple records

Sometimes you want to display many records in the same form. For example, an invoicing application might show all the orders made by a single customer on the same form.

To display multiple records, use a grid control. Grid controls provide a multi-field, multi-record view of data that can make your application's user interface more compelling and effective. They are discussed in "Viewing and editing data with TDBGrid" on page 20-15 and "Creating a grid that contains other data-aware controls" on page 20-28.

**Note** You can't display multiple records when using a unidirectional dataset.

You may want to design a user interface that displays both fields from a single record and grids that represent multiple records. There are two models that combine these two approaches:

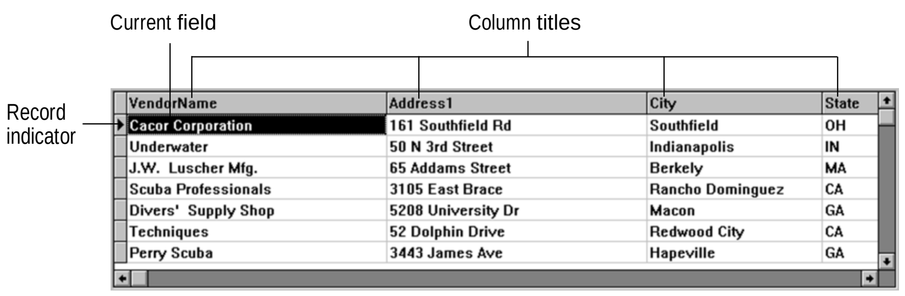
- **Master-detail forms:** You can represent information from both a master table and a detail table by including both controls that display a single field and grid controls. For example, you could display information about a single customer with a detail grid that displays the orders for that customer. For information about linking the underlying tables in a master-detail form, see "Creating master/detail relationships" on page 24-35 and "Establishing master/detail relationships using parameters" on page 24-47.
- **Drill-down forms**: In a form that displays multiple records, you can include single field controls that display detailed information from the current record only. This approach is particularly useful when the records include long memos or graphic information. As the user scrolls through the records of the grid, the memo or graphic updates to represent the value of the current record. Setting this up is very easy. The synchronization between the two displays is automatic if the grid and the memo or image control share a common data source.

**Tip** It is generally not a good idea to combine these two approaches on a single form. It is usually confusing for users to understand the data relationships in such forms.

## Viewing and editing data with TDBGrid

A *TDBGrid* control lets you view and edit records in a dataset in a tabular grid format.

**Figure 20.1** TDBGrid control



| VendorName | Address1 | City | State |
|---|---|---|---|
| Cacor Corporation | 161 Southfield Rd | Southfield | OH |
| Underwater | 50 N 3rd Street | Indianapolis | IN |
| J.W. Luscher Mfg. | 65 Addams Street | Berkely | MA |
| Scuba Professionals | 3105 East Brace | Rancho Dominguez | CA |
| Divers' Supply Shop | 5208 University Dr | Macon | GA |
| Techniques | 52 Dolphin Drive | Redwood City | CA |
| Perry Scuba | 3443 James Ave | Hapeville | GA |

Three factors affect the appearance of records displayed in a grid control:

- Existence of persistent column objects defined for the grid using the Columns editor. Persistent column objects provide great flexibility setting grid and data appearance. For information on using persistent columns, see "Creating a customized grid" on page 20-17.
- Creation of persistent field components for the dataset displayed in the grid. For more information about creating persistent field components using the Fields editor, see Chapter 25, "Working with field components."
- The dataset's *ObjectView* property setting for grids displaying ADT and array fields. See "Displaying ADT and array fields" on page 20-22.

A grid control has a *Columns* property that is itself a wrapper on a *TDBGridColumns* object. *TDBGridColumns* is a collection of *TColumn* objects representing all of the columns in a grid control. You can use the Columns editor to set up column attributes at design time, or use the *Columns* property of the grid to access the properties, events, and methods of *TDBGridColumns* at runtime.

## Using a grid control in its default state

The *State* property of the grid's *Columns* property indicates whether persistent column objects exist for the grid. *Columns.State* is a runtime-only property that is automatically set for a grid. The default state is *csDefault*, meaning that persistent column objects do not exist for the grid. In that case, the display of data in the grid is determined primarily by the properties of the fields in the grid's dataset, or, if there are no persistent field components, by a default set of display characteristics.

When the grid's *Columns.State* property is *csDefault*, grid columns are dynamically generated from the visible fields of the dataset and the order of columns in the grid matches the order of fields in the dataset. Every column in the grid is associated with a field component. Property changes to field components immediately show up in the grid.

Using a grid control with dynamically-generated columns is useful for viewing and editing the contents of arbitrary tables selected at runtime. Because the grid's structure is not set, it can change dynamically to accommodate different datasets. A single grid with dynamically-generated columns can display a Paradox table at one moment, then switch to display the results of an SQL query when the grid's *DataSource* property changes or when the *DataSet* property of the data source itself is changed.

You can change the appearance of a dynamic column at design time or runtime, but what you are actually modifying are the corresponding properties of the field component displayed in the column. Properties of dynamic columns exist only so long as a column is associated with a particular field in a single dataset. For example, changing the *Width* property of a column changes the *DisplayWidth* property of the field associated with that column. Changes made to column properties that are not based on field properties, such as *Font*, exist only for the lifetime of the column.

If a grid's dataset consists of dynamic field components, the fields are destroyed each time the dataset is closed. When the field components are destroyed, all dynamic columns associated with them are destroyed as well. If a grid's dataset consists of persistent field components, the field components exist even when the dataset is closed, so the columns associated with those fields also retain their properties when the dataset is closed.

**Note** Changing a grid's *Columns.State* property to *csDefault* at runtime deletes all column objects in the grid (even persistent columns), and rebuilds dynamic columns based on the visible fields of the grid's dataset.

## Creating a customized grid

A customized grid is one for which you define persistent column objects that describe how a column appears and how the data in the column is displayed. A customized grid lets you configure multiple grids to present different views of the same dataset (different column orders, different field choices, and different column colors and fonts, for example). A customized grid also enables you to let users modify the appearance of the grid at runtime without affecting the fields used by the grid or the field order of the dataset.

Customized grids are best used with datasets whose structure is known at design time. Because they expect field names established at design time to exist in the dataset, customized grids are not well suited to browsing arbitrary tables selected at runtime.

### Understanding persistent columns

When you create persistent column objects for a grid, they are only loosely associated with underlying fields in a grid's dataset. Default property values for persistent columns are dynamically fetched from a default source (the associated field or the grid itself) until a value is assigned to the column property. Until you assign a column property a value, its value changes as its default source changes. Once you assign a value to a column property, it no longer changes when its default source changes.

For example, the default source for a column title caption is an associated field's *DisplayLabel* property. If you modify the *DisplayLabel* property, the column title reflects that change immediately. If you then assign a string to the column title's caption, the tile caption becomes independent of the associated field's *DisplayLabel* property. Subsequent changes to the field's *DisplayLabel* property no longer affect the column's title.

Persistent columns exist independently from field components with which they are associated. In fact, persistent columns do not have to be associated with field objects at all. If a persistent column's *FieldName* property is blank, or if the field name does not match the name of any field in the grid's current dataset, the column's *Field* property is NULL and the column is drawn with blank cells. If you override the cell's default drawing method, you can display your own custom information in the blank cells. For example, you can use a blank column to display aggregated values on the last record of a group of records that the aggregate summarizes. Another possibility is to display a bitmap or bar chart that graphically depicts some aspect of the record's data.

Two or more persistent columns can be associated with the same field in a dataset. For example, you might display a part number field at the left and right extremes of a wide grid to make it easier to find the part number without having to scroll the grid.

**Note** Because persistent columns do not have to be associated with a field in a dataset, and because multiple columns can reference the same field, a customized grid's *FieldCount* property can be less than or equal to the grid's column count. Also note that if the currently selected column in a customized grid is not associated with a field, the grid's *SelectedField* property is NULL and the *SelectedIndex* property is –1.

Persistent columns can be configured to display grid cells as a combo box drop-down list of lookup values from another dataset or from a static pick list, or as an ellipsis button (…) in a cell that can be clicked upon to launch special data viewers or dialogs related to the current cell.

### Creating persistent columns

To customize the appearance of grid at design time, you invoke the Columns editor to create a set of persistent column objects for the grid. At runtime, the *State* property for a grid with persistent column objects is automatically set to *csCustomized*.

To create persistent columns for a grid control,

1. Select the grid component in the form.
2. Invoke the Columns editor by double clicking on the grid's *Columns* property in the Object Inspector.

The Columns list box displays the persistent columns that have been defined for the selected grid. When you first bring up the Columns editor, this list is empty because the grid is in its default state, containing only dynamic columns.

You can create persistent columns for all fields in a dataset at once, or you can create persistent columns on an individual basis. To create persistent columns for all fields:

1. Right-click the grid to invoke the context menu and choose Add All Fields. Note that if the grid is not already associated with a data source, Add All Fields is disabled. Associate the grid with a data source that has an active dataset before choosing Add All Fields.
2. If the grid already contains persistent columns, a dialog box asks if you want to delete the existing columns, or append to the column set. If you choose Yes, any existing persistent column information is removed, and all fields in the current dataset are inserted by field name according to their order in the dataset. If you choose No, any existing persistent column information is retained, and new column information, based on any additional fields in the dataset, are appended to the dataset.
3. Click Close to apply the persistent columns to the grid and close the dialog box.

To create persistent columns individually:

1 Choose the Add button in the Columns editor. The new column will be selected in the list box. The new column is given a sequential number and default name (for example, 0 - TColumn).

2 To associate a field with this new column, set the *FieldName* property in the Object Inspector.

3 To set the title for the new column, expand the *Title* property in the Object Inspector and set its *Caption* property.

4 Close the Columns editor to apply the persistent columns to the grid and close the dialog box.

At runtime, you can switch to persistent columns by assigning *csCustomized* to the *Columns.State* property. Any existing columns in the grid are destroyed and new persistent columns are built for each field in the grid's dataset. You can then add a persistent column at runtime by calling the *Add* method for the column list:

```
DBGrid1.Columns.Add;
```

## Deleting persistent columns

Deleting a persistent column from a grid is useful for eliminating fields that you do not want to display. To remove a persistent column from a grid,

1 Double-click the grid to display the Columns editor.

2 Select the field to remove in the Columns list box.

3 Click Delete (you can also use the context menu or Del key, to remove a column).

**Note** If you delete all the columns from a grid, the *Columns.State* property reverts to its *csDefault* state and automatically build dynamic columns for each field in the dataset.

You can delete a persistent column at runtime by simply freeing the column object:

```
DBGrid1.Columns[5].Free;
```

## Arranging the order of persistent columns

The order in which columns appear in the Columns editor is the same as the order the columns appear in the grid. You can change the column order by dragging and dropping columns within the Columns list box.

To change the order of a column,

1 Select the column in the Columns list box.

2 Drag it to a new location in the list box.

You can also change the column order at runtime by clicking on the column title and dragging the column to a new position.

**Note** Reordering persistent fields in the Fields editor also reorders columns in a default grid, but not a custom grid.

**Important** You cannot reorder columns in grids containing both dynamic columns and dynamic fields at design time, since there is nothing persistent to record the altered field or column order.

At runtime, a user can use the mouse to drag a column to a new location in the grid if its *DragMode* property is set to *dmManual*. Reordering the columns of a grid with a *State* property of *csDefault* state also reorders field components in the dataset underlying the grid. The order of fields in the physical table is not affected. To prevent a user from rearranging columns at runtime, set the grid's *DragMode* property to *dmAutomatic*.

At runtime, the grid's *OnColumnMoved* event fires after a column has been moved.

## Setting column properties at design time

Column properties determine how data is displayed in the cells of that column. Most column properties obtain their default values from properties associated with another component (called the *default source*) such as a grid or an associated field component.

To set a column's properties, select the column in The Columns editor and set its properties in the Object Inspector. The following table summarizes key column properties you can set.

**Table 20.2** Column properties

| Property | Purpose |
|---|---|
| Alignment | Left justifies, right justifies, or centers the field data in the column. Default source: *TField.Alignment*. |
| ButtonStyle | *cbsAuto*: (default) Displays a drop-down list if the associated field is a lookup field, or if the column's *PickList* property contains data. |
| | *cbsEllipsis*: Displays an ellipsis (...) button to the right of the cell. Clicking on the button fires the grid's *OnEditButtonClick* event. |
| | *cbsNone*: The column uses only the normal edit control to edit data in the column. |
| Color | Specifies the background color of the cells of the column. Default source: *TDBGrid.Color. (*For text foreground color, see the Font property.) |
| DropDownRows | The number of lines of text displayed by the drop-down list. Default: 7. |
| Expanded | Specifies whether the column is expanded. Only applies to columns representing ADT or array fields. |
| FieldName | Specifies the field name associated with this column. This can be blank. |
| ReadOnly | *True*: The data in the column cannot be edited by the user. |
| | *False*: (default) The data in the column can be edited. |

**Table 20.2** Column properties (continued)

| Property | Purpose |
|---|---|
| Width | Specifies the width of the column in screen pixels. Default source: *TField.DisplayWidth*. |
| Font | Specifies the font type, size, and color used to draw text in the column. Default source: *TDBGrid.Font*. |
| PickList | Contains a list of values to display in a drop-down list in the column. |
| Title | Sets properties for the title of the selected column. |

The following table summarizes the options you can specify for the *Title* property.

**Table 20.3** Expanded TColumn Title properties

| Property | Purpose |
|---|---|
| Alignment | Left justifies (default), right justifies, or centers the caption text in the column title. |
| Caption | Specifies the text to display in the column title. Default source: *TField.DisplayLabel*. |
| Color | Specifies the background color used to draw the column title cell. Default source: *TDBGrid.FixedColor*. |
| Font | Specifies the font type, size, and color used to draw text in the column title. Default source: *TDBGrid.TitleFont*. |

## Defining a lookup list column

You can create a column that displays a drop-down list of values, similar to a lookup combo box control. To specify that the column acts like a combo box, set the column's *ButtonStyle* property to *cbsAuto*. Once you populate the list with values, the grid automatically displays a combo box-like drop-down button when a cell of that column is in edit mode.

There are two ways to populate that list with the values for users to select:

- You can fetch the values from a lookup table. To make a column display a drop-down list of values drawn from a separate lookup table, you must define a lookup field in the dataset. For information about creating lookup fields, see "Defining a lookup field" on page 25-9. Once the lookup field is defined, set the column's *FieldName* to the lookup field name. The drop-down list is automatically populated with lookup values defined by the lookup field.
- You can specify a list of values explicitly at design time. To enter the list values at design time, double-click the *PickList* property for the column in the Object Inspector. This brings up the String List editor, where you can enter the values that populate the pick list for the column.

By default, the drop-down list displays 7 values. You can change the length of this list by setting the *DropDownRows* property.

**Note** To restore a column with an explicit pick list to its normal behavior, delete all the text from the pick list using the String List editor.

### Putting a button in a column

A column can display an ellipsis button (…) to the right of the normal cell editor. Ctrl+Enter or a mouse click fires the grid's *OnEditButtonClick* event. You can use the ellipsis button to bring up forms containing more detailed views of the data in the column. For example, in a table that displays summaries of invoices, you could set up an ellipsis button in the invoice total column to bring up a form that displays the items in that invoice, or the tax calculation method, and so on. For graphic fields, you could use the ellipsis button to bring up a form that displays an image.

To create an ellipsis button in a column:

1. Select the column in the *Columns* list box.
2. Set *ButtonStyle* to *cbsEllipsis*.
3. Write an *OnEditButtonClick* event handler.

### Restoring default values to a column

At runtime you can test a column's *AssignedValues* property to determine whether a column property has been explicitly assigned. Values that are not explicitly defined are dynamically based on the associated field or the grid's defaults.

You can undo property changes made to one or more columns. In the Columns editor, select the column or columns to restore, and then select Restore Defaults from the context menu. Restore defaults discards assigned property settings and restores a column's properties to those derived from its underlying field component

At runtime, you can reset all default properties for a single column by calling the column's *RestoreDefaults* method. You can also reset default properties for all columns in a grid by calling the column list's *RestoreDefaults* method:

```
DBGrid1.Columns.RestoreDefaults;
```

## Displaying ADT and array fields

Sometimes the fields of the grid's dataset do not represent simple values such as text, graphics, numerical values, and so on. Some database servers allow fields that are a composite of simpler data types, such as ADT fields or array fields.

There are two ways a grid can display composite fields:

- It can "flatten out" the field so that each of the simpler types that make up the field appears as a separate field in the dataset. When a composite field is flattened out, its constituents appear as separate fields that reflect their common source only in that each field name is preceded by the name of the common parent field in the underlying database table.

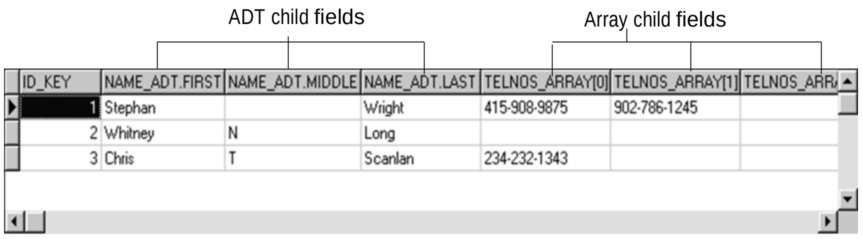
  To display composite fields as if they were flattened out, set the dataset's *ObjectView* property to *False*. The dataset stores composite fields as a set of separate fields, and the grid reflects this by assigning each constituent part a separate column.

- It can display composite fields in a single column, reflecting the fact that they are a single field. When displaying composite fields in a single column, the column can be expanded and collapsed by clicking on the arrow in the title bar of the field, or by setting the *Expanded* property of the column:
  - When a column is expanded, each child field appears in its own sub-column with a title bar that appears below the title bar of the parent field. That is, the title bar for the grid increases in height, with the first row giving the name of the composite field, and the second row subdividing that for the individual parts. Fields that are not composites appear with title bars that are extra high. This expansion continues for constituents that are in turn composite fields (for example, a detail table nested in a detail table), with the title bar growing in height accordingly.
  - When the field is collapsed, only one column appears with an uneditable comma delimited string containing the child fields.

  To display a composite field in an expanding and collapsing column, set the dataset's *ObjectView* property to *True*. The dataset stores the composite field as a single field component that contains a set of nested sub-fields. The grid reflects this in a column that can expand or collapse

Figure 20.2 shows a grid with an ADT field and an array field. The dataset's *ObjectView* property is set to *False* so that each child field has a column.

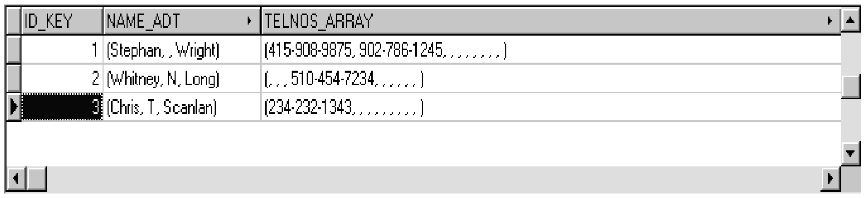
**Figure 20.2** TDBGrid control with ObjectView set to False



| ID_KEY | NAME_ADT.FIRST | NAME_ADT.MIDDLE | NAME_ADT.LAST | TELNOS_ARRAY[0] | TELNOS_ARRAY[1] | TELNOS_ARR |
|---|---|---|---|---|---|---|
| 1 | Stephan |  | Wright | 415-908-9875 | 902-786-1245 |  |
| 2 | Whitney | N | Long |  |  |  |
| 3 | Chris | T | Scanlan | 234-232-1343 |  |  |

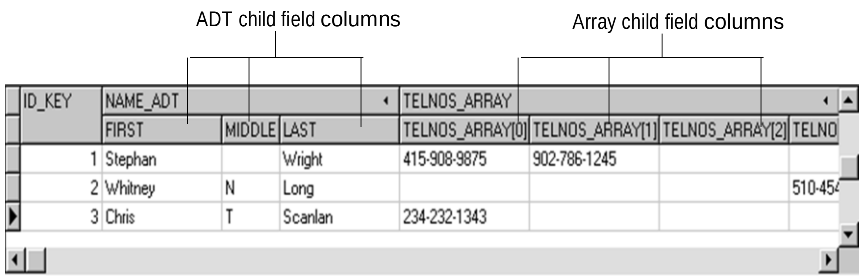
Figure 20.3 and 20.4 show the grid with an ADT field and an array field. Figure 20.3 shows the fields collapsed. In this state they cannot be edited. Figure 20.4 shows the fields expanded. The fields are expanded and collapsed by clicking on the arrow in the fields title bar.

**Figure 20.3** TDBGrid control with Expanded set to False



| ID_KEY | NAME_ADT | TELNOS_ARRAY |
|---|---|---|
| 1 | (Stephan, , Wright) | (415-908-9875, 902-786-1245, , , , , , , , ) |
| 2 | (Whitney, N, Long) | (, , , 510-454-7234, , , , , , ) |
| 3 | (Chris, T, Scanlan) | (234-232-1343, , , , , , , , , ) |

**Figure 20.4** TDBGrid control with Expanded set to True



| ID_KEY | NAME_ADT | | | TELNOS_ARRAY | | | |
|---|---|---|---|---|---|---|---|
| | FIRST | MIDDLE | LAST | TELNOS_ARRAY[0] | TELNOS_ARRAY[1] | TELNOS_ARRAY[2] | TELNO |
| 1 | Stephan | | Wright | 415-908-9875 | 902-786-1245 | | |
| 2 | Whitney | N | Long | | | | 510-454 |
| 3 | Chris | T | Scanlan | 234-232-1343 | | | |

The following table lists the properties that affect the way ADT and array fields appear in a *TDBGrid*:

**Table 20.4** Properties that affect the way composite fields appear

| Property | Object | Purpose |
|---|---|---|
| Expandable | TColumn | Indicates whether the column can be expanded to show child fields in separate, editable columns. (read-only) |
| Expanded | TColumn | Specifies whether the column is expanded. |
| MaxTitleRows | TDBGrid | Specifies the maximum number of title rows that can appear in the grid |
| ObjectView | TDataSet | Specifies whether fields are displayed flattened out, or in object mode, where each object field can be expanded and collapsed. |
| ParentColumn | TColumn | Refers to the TColumn object that owns the child field's column. |

**Note** In addition to ADT and array fields, some datasets include fields that refer to another dataset (dataset fields) or a record in another dataset (reference) fields. Data-aware grids display such fields as "(DataSet)" or "(Reference)", respectively. At runtime an ellipsis button appears to the right. Clicking on the ellipsis brings up a new form with a grid displaying the contents of the field. For dataset fields, this grid displays the dataset that is the field's value. For reference fields, this grid contains a single row that displays the record from another dataset.

## Setting grid options

You can use the grid *Options* property at design time to control basic grid behavior and appearance at runtime. When a grid component is first placed on a form at design time, the *Options* property in the Object Inspector is displayed with a + (plus) sign to indicate that the *Options* property can be expanded to display a series of Boolean properties that you can set individually. To view and set these properties, click on the + sign. The list of options in the Object Inspector below the *Options* property. The + sign changes to a – (minus) sign, that collapses the list back when you click it.

The following table lists the *Options* properties that can be set, and describes how they affect the grid at runtime.

**Table 20.5** Expanded TDBGrid Options properties

| Option | Purpose |
|---|---|
| dgEditing | *True*: (Default). Enables editing, inserting, and deleting records in the grid. |
| | *False*: Disables editing, inserting, and deleting records in the grid. |
| dgAlwaysShowEditor | *True*: When a field is selected, it is in Edit state. |
| | *False*: (Default). A field is not automatically in Edit state when selected. |
| dgTitles | *True*: (Default). Displays field names across the top of the grid. |
| | *False*: Field name display is turned off. |
| dgIndicator | *True*: (Default). The indicator column is displayed at the left of the grid, and the current record indicator (an arrow at the left of the grid) is activated to show the current record. On insert, the arrow becomes an asterisk. On edit, the arrow becomes an I-beam. |
| | *False*: The indicator column is turned off. |
| dgColumnResize | *True*: (Default). Columns can be resized by dragging the column rulers in the title area. Resizing changes the corresponding width of the underlying *TField* component. |
| | *False*: Columns cannot be resized in the grid. |
| dgColLines | *True*: (Default). Displays vertical dividing lines between columns. |
| | *False*: Does not display dividing lines between columns. |
| dgRowLines | *True*: (Default). Displays horizontal dividing lines between records. |
| | *False*: Does not display dividing lines between records. |
| dgTabs | *True*: (Default). Enables tabbing between fields in records. |
| | *False*: Tabbing exits the grid control. |
| dgRowSelect | *True*: The selection bar spans the entire width of the grid. |
| | *False*: (Default). Selecting a field in a record selects only that field. |
| dgAlwaysShowSelection | *True*: (Default). The selection bar in the grid is always visible, even if another control has focus. |
| | *False*: The selection bar in the grid is only visible when the grid has focus. |
| dgConfirmDelete | *True*: (Default). Prompt for confirmation to delete records (Ctrl+Del). |
| | *False*: Delete records without confirmation. |
| dgCancelOnExit | *True*: (Default). Cancels a pending insert when focus leaves the grid. This option prevents inadvertent posting of partial or blank records. |
| | *False*: Permits pending inserts. |
| dgMultiSelect | *True*: Allows user to select noncontiguous rows in the grid using Ctrl+Shift or Shift+ arrow keys. |
| | *False*: (Default). Does not allow user to multi-select rows. |

## Editing in the grid

At runtime, you can use a grid to modify existing data and enter new records, if the following default conditions are met:

- The *CanModify* property of the *Dataset* is *True*.
- The *ReadOnly* property of grid is *False*.

When a user edits a record in the grid, changes to each field are posted to an internal record buffer, but are not posted until the user moves to a different record in the grid. Even if focus is changed to another control on a form, the grid does not post changes until another the cursor for the dataset is moved to another record. When a record is posted, the dataset checks all associated data-aware components for a change in status. If there is a problem updating any fields that contain modified data, the grid raises an exception, and does not modify the record.

**Note** If your application caches updates, posting record changes only adds them to an internal cache. They are not posted back to the underlying database table until your application applies the updates.

You can cancel all edits for a record by pressing Esc in any field before moving to another record.

## Controlling grid drawing

Your first level of control over how a grid control draws itself is setting column properties. The grid automatically uses the font, color, and alignment properties of a column to draw the cells of that column. The text of data fields is drawn using the *DisplayFormat* or *EditFormat* properties of the field component associated with the column.

You can augment the default grid display logic with code in a grid's *OnDrawColumnCell* event. If the grid's *DefaultDrawing* property is *True*, all the normal drawing is performed before your *OnDrawColumnCell* event handler is called. Your code can then draw on top of the default display. This is primarily useful when you have defined a blank persistent column and want to draw special graphics in that column's cells.

If you want to replace the drawing logic of the grid entirely, set *DefaultDrawing* to *False* and place your drawing code in the grid's *OnDrawColumnCell* event. If you want to replace the drawing logic only in certain columns or for certain field data types, you can call the *DefaultDrawColumnCell* inside your *OnDrawColumnCell* event handler to have the grid use its normal drawing code for selected columns. This reduces the amount of work you have to do if you only want to change the way Boolean field types are drawn, for example.

## Responding to user actions at runtime

You can modify grid behavior by writing event handlers to respond to specific actions within the grid at runtime. Because a grid typically displays many fields and records at once, you may have very specific needs to respond to changes to individual columns. For example, you might want to activate and deactivate a button elsewhere on the form every time a user enters and exits a specific column.

The following table lists the grid events available in the Object Inspector.

**Table 20.6** Grid control events

| Event | Purpose |
|---|---|
| OnCellClick | Occurs when a user clicks on a cell in the grid. |
| OnColEnter | Occurs when a user moves into a column on the grid. |
| OnColExit | Occurs when a user leaves a column on the grid. |
| OnColumnMoved | Occurs when the user moves a column to a new location. |
| OnDblClick | Occurs when a user double clicks in the grid. |
| OnDragDrop | Occurs when a user drags and drops in the grid. |
| OnDragOver | Occurs when a user drags over the grid. |
| OnDrawColumnCell | Occurs when application needs to draw individual cells. |
| OnDrawDataCell | (obsolete) Occurs when application needs to draw individual cells if *State* is *csDefault*. |
| OnEditButtonClick | Occurs when the user clicks on an ellipsis button in a column. |
| OnEndDrag | Occurs when a user stops dragging on the grid. |
| OnEnter | Occurs when the grid gets focus. |
| OnExit | Occurs when the grid loses focus. |
| OnKeyDown | Occurs when a user presses any key or key combination on the keyboard when in the grid. |
| OnKeyPress | Occurs when a user presses a single alphanumeric key on the keyboard when in the grid. |
| OnKeyUp | Occurs when a user releases a key when in the grid. |
| OnStartDrag | Occurs when a user starts dragging on the grid. |
| OnTitleClick | Occurs when a user clicks the title for a column. |

There are many uses for these events. For example, you might write a handler for the *OnDblClick* event that pops up a list from which a user can choose a value to enter in a column. Such a handler would use the *SelectedField* property to determine to current row and column.

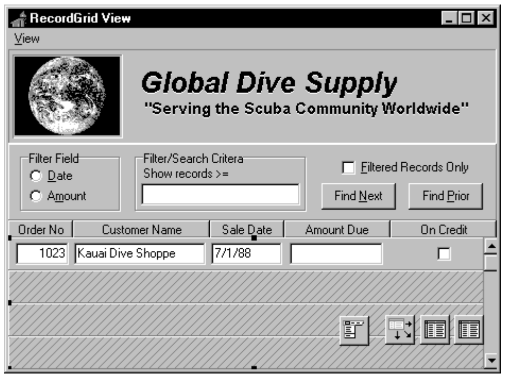
## Creating a grid that contains other data-aware controls

A *TDBCtrlGrid* control displays multiple fields in multiple records in a tabular grid format. Each cell in a grid displays multiple fields from a single row. To use a database control grid:

1. Place a database control grid on a form.
2. Set the grid's *DataSource* property to the name of a data source.
3. Place individual data controls within the design cell for the grid. The design cell for the grid is the top or leftmost cell in the grid, and is the only cell into which you can place other controls.
4. Set the *DataField* property for each data control to the name of a field. The data source for these data controls is already set to the data source of the database control grid.
5. Arrange the controls within the cell as desired.

When you compile and run an application containing a database control grid, the arrangement of data controls you set in the design cell at runtime is replicated in each cell of the grid. Each cell displays a different record in a dataset.

**Figure 20.5** TDBCtrlGrid at design time



The following table summarizes some of the unique properties for database control grids that you can set at design time:

**Table 20.7** Selected database control grid properties

| Property | Purpose |
|---|---|
| AllowDelete | *True* (default): Permits record deletion. |
| | *False*: Prevents record deletion. |
| AllowInsert | *True* (default): Permits record insertion. |
| | *False*: Prevents record insertion. |
| ColCount | Sets the number of columns in the grid. Default = 1. |
| Orientation | *goVertical* (default): Display records from top to bottom. |
| | *goHorizontal*: Displays records from left to right. |
| PanelHeight | Sets the height for an individual panel. Default = 72. |
| PanelWidth | Sets the width for an individual panel. Default = 200. |
| RowCount | Sets the number of panels to display. Default = 3. |
| ShowFocus | *True* (default): Displays a focus rectangle around the current record's panel at runtime. |
| | *False*: Does not display a focus rectangle. |

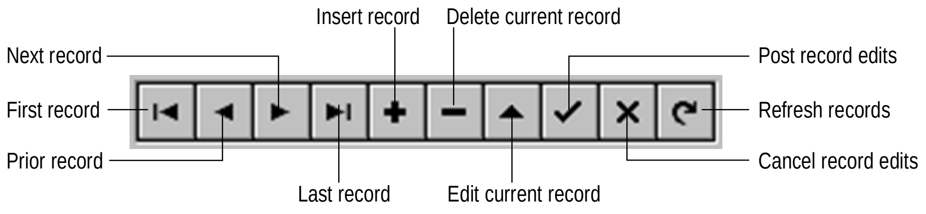
For more information about database control grid properties and methods, see the online *VCL Reference*.

## Navigating and manipulating records

*TDBNavigator* provides users a simple control for navigating through records in a dataset, and for manipulating records. The navigator consists of a series of buttons that enable a user to scroll forward or backward through records one at a time, go to the first record, go to the last record, insert a new record, update an existing record, post data changes, cancel data changes, delete a record, and refresh record display.

Figure 20.6 shows the navigator that appears by default when you place it on a form at design time. The navigator consists of a series of buttons that let a user navigate from one record to another in a dataset, and edit, delete, insert, and post records. The *VisibleButtons* property of the navigator enables you to hide or show a subset of these buttons dynamically.

**Figure 20.6** Buttons on the TDBNavigator control



The following table describes the buttons on the navigator.

**Table 20.8** TDBNavigator buttons

| Button | Purpose |
|---|---|
| First | Calls the dataset's *First* method to set the current record to the first record. |
| Prior | Calls the dataset's *Prior* method to set the current record to the previous record. |
| Next | Calls the dataset's *Next* method to set the current record to the next record. |
| Last | Calls the dataset's *Last* method to set the current record to the last record. |
| Insert | Calls the dataset's *Insert* method to insert a new record before the current record, and set the dataset in Insert state. |
| Delete | Deletes the current record. If the *ConfirmDelete* property is *True* it prompts for confirmation before deleting. |
| Edit | Puts the dataset in Edit state so that the current record can be modified. |
| Post | Writes changes in the current record to the database. |
| Cancel | Cancels edits to the current record, and returns the dataset to Browse state. |
| Refresh | Clears data control display buffers, then refreshes its buffers from the physical table or query. Useful if the underlying data may have been changed by another application. |

## Choosing navigator buttons to display

When you first place a *TDBNavigator* on a form at design time, all its buttons are visible. You can use the *VisibleButtons* property to turn off buttons you do not want to use on a form. For example, when working with a unidirectional dataset, only the *First*, *Next*, and *Refresh* buttons are meaningful. On a form that is intended for browsing rather than editing, you might want to disable the *Edit*, *Insert*, *Delete*, *Post*, and *Cancel* buttons.

### Hiding and showing navigator buttons at design time

The *VisibleButtons* property in the Object Inspector is displayed with a + sign to indicate that it can be expanded to display a Boolean value for each button on the navigator. To view and set these values, click on the + sign. The list of buttons that can be turned on or off appears in the Object Inspector below the *VisibleButtons* property. The + sign changes to a – (minus) sign, which you can click to collapse the list of properties.

Button visibility is indicated by the *Boolean* state of the button value. If a value is set to *True*, the button appears in the *TDBNavigator*. If ***False***, the button is removed from the navigator at design time and runtime.

**Note** As button values are set to *False*, they are removed from the *TDBNavigator* on the form, and the remaining buttons are expanded in width to fill the control. You can drag the control's handles to resize the buttons.

### Hiding and showing navigator buttons at runtime

At runtime you can hide or show navigator buttons in response to user actions or application states. For example, suppose you provide a single navigator for navigating through two different datasets, one of which permits users to edit records, and the other of which is read-only. When you switch between datasets, you want to hide the navigator's *Insert, Delete, Edit, Post, Cancel*, and *Refresh* buttons for the read-only dataset, and show them for the other dataset.

For example, suppose you want to prevent edits to the *OrdersTable* by hiding the *Insert, Delete, Edit, Post, Cancel,* and *Refresh* buttons on the navigator, but that you also want to allow editing for the *CustomersTable*. The *VisibleButtons* property controls which buttons are displayed in the navigator. Here's one way you might code the event handler:

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
  begin
    DBNavigatorAll.DataSource := CustomerCompany.DataSource;
    DBNavigatorAll.VisibleButtons := [nbFirst,nbPrior,nbNext,nbLast];
  end
  else
  begin
    DBNavigatorAll.DataSource := OrderNum.DataSource;
    DBNavigatorAll.VisibleButtons := DBNavigatorAll.VisibleButtons + [nbInsert,
      nbDelete,nbEdit,nbPost,nbCancel,nbRefresh];
  end;
end;
```

## Displaying fly-over help

To display fly-over help for each navigator button at runtime, set the navigator *ShowHint* property to *True*. When *ShowHint* is *True*, the navigator displays fly-by Help hints whenever you pass the mouse cursor over the navigator buttons. *ShowHint* is *False* by default.

The *Hints* property controls the fly-over help text for each button. By default *Hints* is an empty string list. When *Hints* is empty, each navigator button displays default help text. To provide customized fly-over help for the navigator buttons, use the String list editor to enter a separate line of hint text for each button in the *Hints* property. When present, the strings you provide override the default hints provided by the navigator control.

## Using a single navigator for multiple datasets

As with other data-aware controls, a navigator's *DataSource* property specifies the data source that links the control to a dataset. By changing a navigator's *DataSource* property at runtime, a single navigator can provide record navigation and manipulation for multiple datasets.

Suppose a form contains two edit controls linked to the *CustomersTable* and *OrdersTable* datasets through the *CustomersSource* and *OrdersSource* data sources respectively. When a user enters the edit control connected to *CustomersSource*, the navigator should also use *CustomersSource*, and when the user enters the edit control connected to *OrdersSource*, the navigator should switch to *OrdersSource* as well. You can code an *OnEnter* event handler for one of the edit controls, and then share that event with the other edit control. For example:

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
    DBNavigatorAll.DataSource := CustomerCompany.DataSource
  else
    DBNavigatorAll.DataSource := OrderNum.DataSource;
end;
```

Chapter

21

# Creating reports with Rave Reports

This chapter provides an overview of using Rave Reports from Nevrona Designs to generate reports within a Delphi application. Additional documentation for Rave Reports is included in the Delphi installation directory, as described in "Getting more information" on page 21-6.

**Note:** Rave Reports is automatically installed with the Professional and Enterprise editions of Delphi.

## Overview

Rave Reports is a component-based visual report design tool that simplifies the process of adding reports to an application. You can use Rave Reports to create a variety of reports, from simple banded reports to more complex, highly customized reports. Report features include:

- Word wrapped memos
- Full graphics
- Justification
- Precise page positioning
- Printer configuration
- Font control
- Print preview
- Reuse of report content
- PDF, HTML, RTF, and text report renditions

## Getting started

You can use Rave Reports in both VCL and CLX applications to generate reports from database and non-database data. The following procedure explains how to add a simple report to an existing database application.

1 Open a database application in Delphi.

2 From the Rave page of the Component palette, add the TRvDataSetConnection component to a form in the application.

3 In the Object Inspector, set the *DataSet* property to a dataset component that is already defined in your application.

4 Use the Rave Visual Designer to design your report and create a report project file (.rav file).

   a Choose Tools|Rave Designer to launch the Rave Visual Designer.

   b Choose File|New Data Object to display the Data Connections dialog box.

   c In the Data Object Type list, select Direct Data View and click Next.

   d In the Active Data Connections list, select RVDataSetConnection1 and click Finish.

   In the Project Tree on the left side of the Rave Visual Designer window, expand the Data View Dictionary node, then expand the newly created DataView1 node. Your application data fields are displayed under the DataView1 node.

   e Choose Tools|Report Wizards|Simple Table to display the Simple Table wizard.

   f Select DataView1 and click Next.

   g Select two or three fields that you want to display in the report and click Next.

   h Follow the prompts on the subsequent wizard pages to set the order of the fields, margins, heading text, and fonts to be used in the report.

   i On the final wizard page, click Generate to complete the wizard and display the report in the Page Designer.

   j Choose File|Save as to display the Save As dialog box. Navigate to the directory in which your Delphi application is located and save the Rave project file as MyRave.rav.

   k Minimize the Rave Visual Designer window and return to Delphi.

5 From the Rave page of the Component palette, add the Rave project component, TRvProject, to the form.

6 In the Object Inspector, set the *ProjectFile* property to the report project file (MyRave.rav) that you created in step j.

**7** From the Standard page of the Component palette, add the TButton component.

**8** In the Object Inspector, click the Events tab and double-click the OnClick event.

**9** Write an event handler that uses the ExecuteReport method to execute the Rave project component.

**10** Press F9 to run the application.

**11** Click the button that you added in step 7.

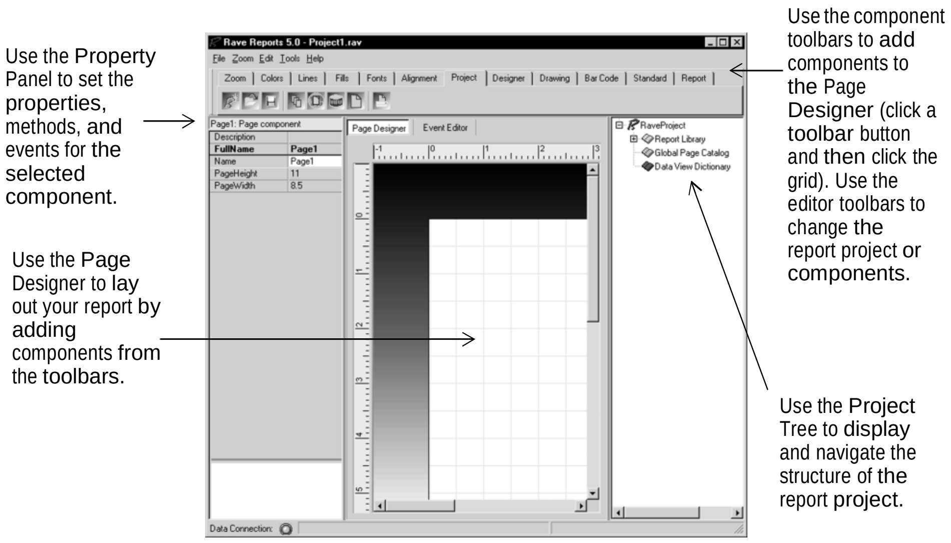
**12** The Output Options dialog box is displayed. Click OK to display the report.

For a more information on using the Rave Visual Designer, use the Help menu or see the Rave Reports documentation listed in “Getting more information” on page 21-6.

## The Rave Visual Designer

To launch the Rave Visual Designer, do one of the following:

- Choose Tools | Rave Designer.
- Double-click a TRvProject component on a form.
- Right-click a TRvProject component on a form, and choose Rave Visual Designer.


Rave Reports 5.0 - Project1.rav
File Zoom Edit Tools Help
Zoom | Colors | Lines | Fills | Fonts | Alignment | Project | Designer | Drawing | Bar Code | Standard | Report
Page1: Page component
Page Designer
Event Editor
RaveProject
Report Library
Global Page Catalog
Data View Dictionary
Data Connection:

| | |
|---|---|
| Description | |
| FullName | Page1 |
| Name | Page1 |
| PageHeight | 11 |
| PageWidth | 8.5 |

Use the Property Panel to set the properties, methods, and events for the selected component.

Use the Page Designer to lay out your report by adding components from the toolbars.

Use the component toolbars to add components to the Page Designer (click a toolbar button and then click the grid). Use the editor toolbars to change the report project or components.

Use the Project Tree to display and navigate the structure of the report project.

For a detailed information on using the Rave Visual Designer, use the Help menu or see the Rave Reports documentation listed in “Getting more information” on page 21-6.

## Component overview

This section provides an overview of the Rave Reports components. For detailed component information, see the documentation listed in "Getting more information" on page 21-6.

### VCL/CLX components

The VCL/CLX components are non-visual components that you add to a form in your VCL or CLX application. They are available on the Rave page of the Component palette. There are four categories of components: engine, render, data connection and Rave project.

#### Engine components

The Engine components are used to generate reports. Reports can be generated from a pre-defined visual definition (using the *Engine* property of TRvProject) or by making calls to the Rave code-based API library from within the OnPrint event. The engine components are:

TRvNDRWriter
TRvSystem

#### Render components

The Render components are used to convert an NDR file (Rave snapshot report file) or a stream generated from TRvNDRWriter to a variety of formats. Rendering can be done programmatically or added to the standard setup and preview dialogs of TRvSystem by dropping a render component on an active form or data module within your application. The render components are:

| | |
|---|---|
| TRvRenderPreview | TRvRenderPrinter |
| TRvRenderPDF | TRvRenderHTML |
| TRvRenderRTF | TRvRenderText |

#### Data connection components

The Data Connection components provide the link between application data and the Direct Data Views in visually designed Rave reports. The data connection components are:

| | |
|---|---|
| TRvCustomConnection | TRvDataSetConnection |
| TRvTableConnection | TRvQueryConnection |

### Rave project component

The TRvProject component interface with and executes visually designed Rave reports within an application. Normally a TRvSystem component would be assigned to the *Engine* property. The reporting project (.rav) should be specified in the *ProjectFile* property or loaded into the DFM using the *StoreRAV* property. Project parameters can be set using the SetParam method and reports can be executed using the ExecuteReport method.

## Reporting components

The following components are available in the Rave Visual Designer.

### Project components

The Project toolbar provides the essential building blocks for all reports. The project components are:

TRavePage
TRaveProjectManager
TRaveReport

### Data objects

Data objects connect to data or control access to reports from the Rave Reporting Server. The File|New Data Object menu command displays the Data Connections dialog box, which you can use to create each of the data objects. The data object components are:

| | | |
|---|---|---|
| TRaveDatabase | TRaveDriverDataView | TRaveSimpleSecurity |
| TRaveDirectDataView | TRaveLookupSecurity | |

### Standard components

The Standard toolbar provides components that are frequently used when designing reports. The standard components are:

| | | |
|---|---|---|
| TRaveBitmap | TRaveMetaFile | TRaveText |
| TRaveFontMaster | TRavePageNumInit | |
| TRaveMemo | TRaveSection | |

### Drawing components

The Drawing toolbar provides components to create lines and shapes in a report. To color and style the components, use the Fills, Lines, and Colors toolbars. The drawing components are:

| | | |
|---|---|---|
| TRaveCircle | TRaveLine | TRaveVLine |
| TRaveEllipse | TRaveRectangle | |
| TRaveHLine | TRaveSquare | |

### Report components

The Report toolbar provides components that used most often in data-aware reports. The report components are:

| | | |
|---|---|---|
| Band Style Editor | TRaveCalcText | TRaveDataMirrorSection |
| DataText Editor | TRaveCalcTotal | TRaveDataText |
| TRaveBand | TRaveDataBand | TRaveRegion |
| TRaveCalcController | TRaveDataCycle | |
| TRaveCalcOp Component | TRaveDataMemo | |

### Bar code components

The Bar Code toolbar provides different types of bar codes in a report. The bar code components are:

| | | |
|---|---|---|
| TRaveCode128BarCode | TRaveEANBarCode | TRavePostNetBarCode |
| TRaveCode39BarCode | TRaveI2of5Bar Code | TRaveUPCBarCode |

## Getting more information

Delphi includes the following Nevrona Designs documentation for Rave Reports.

**Table 21.1** Rave Reports documentation

| Title | Description |
|---|---|
| *Rave Visual Designer Manual for Reference and Learning* | Provides detailed information about using the Rave Visual Designer to create reports. |
| *Rave Tutorial and Reference* | Provides step-by-step instructions on using the Rave Reports components and includes a reference of classes, components, and units. |
| *Rave Application Interface Technology Specification* | Explains how to create custom Rave Reports components, property editors, component editors, project editors, and control the Rave environment. |

These books are distributed as PDF files on the Delphi installation CD.

Most of the information in the PDF files is also available in the online Help. To display online Help for a Rave Reports component on a form, select the component and press F1. To display online Help for the Rave Visual Designer, use the Help menu.

Chapter

22

# Using decision support components

The decision support components help you create cross-tabulated—or, crosstab—tables and graphs. You can then use these tables and graphs to view and summarize data from different perspectives. For more information on cross-tabulated data, see "About crosstabs" on page 22-2.

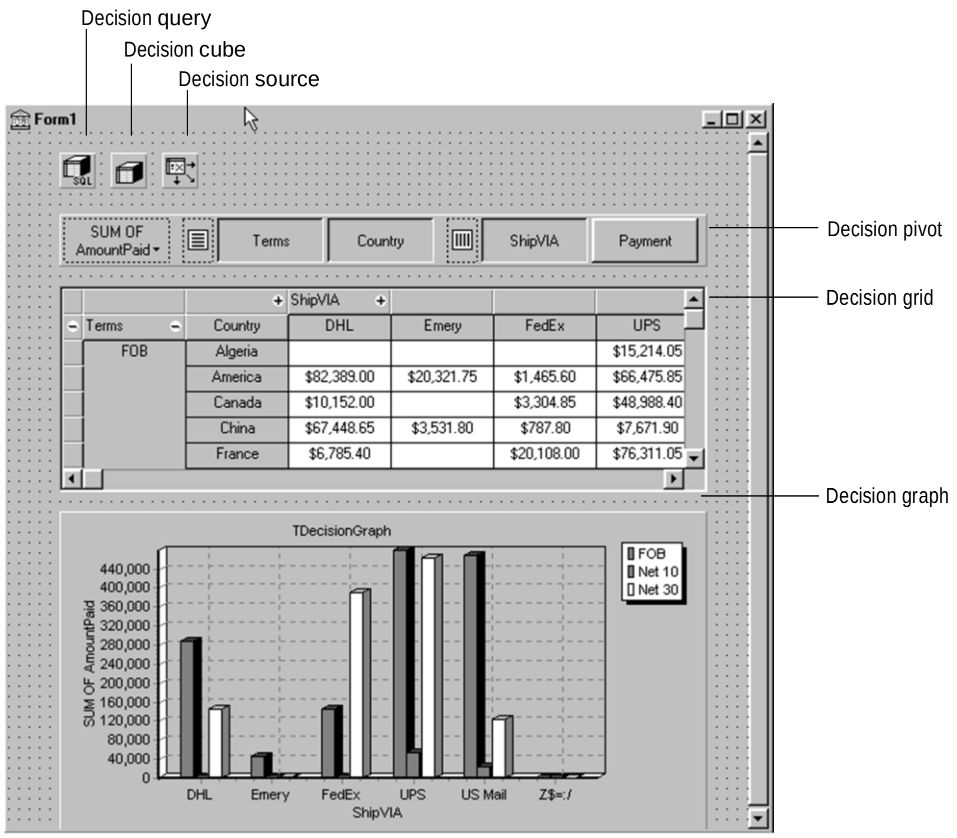
## Overview

The decision support components appear on the Decision Cube page of the Component palette:

- The decision cube, *TDecisionCube*, is a multidimensional data store.
- The decision source, *TDecisionSource*, defines the current pivot state of a decision grid or a decision graph.
- The decision query, *TDecisionQuery*, is a specialized form of *TQuery* used to define the data in a decision cube.
- The decision pivot, *TDecisionPivot*, lets you open or close decision cube dimensions, or fields, by pressing buttons.
- The decision grid, *TDecisionGrid*, displays single- and multidimensional data in table form.
- The decision graph, *TDecisionGraph*, displays fields from a decision grid as a dynamic graph that changes when data dimensions are modified.

Figure 22.1 shows all the decision support components placed on a form at design time.

**Figure 22.1** Decision support components at design time



|  |  | ShipVIA |  |  |  |
|---|---|---|---|---|---|
| Terms | Country | DHL | Emery | FedEx | UPS |
| FOB | Algeria |  |  |  | $15,214.05 |
|  | America | $82,389.00 | $20,321.75 | $1,465.60 | $66,475.85 |
|  | Canada | $10,152.00 |  | $3,304.85 | $48,988.40 |
|  | China | $67,448.65 | $3,531.80 | $787.80 | $7,671.90 |
|  | France | $6,785.40 |  | $20,108.00 | $76,311.05 |

## About crosstabs

Cross-tabulations, or crosstabs, are a way of presenting subsets of data so that relationships and trends are more visible. Table fields become the dimensions of the crosstab while field values define categories and summaries within a dimension.

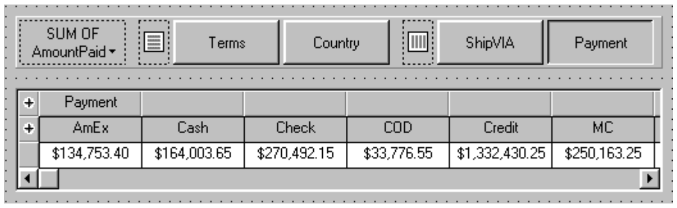
You can use the decision support components to set up crosstabs in forms. *TDecisionGrid* shows data in a table, while *TDecisionGraph* charts it graphically. *TDecisionPivot* has buttons that make it easier to display and hide dimensions and move them between columns and rows.

Crosstabs can be one-dimensional or multidimensional.

## One-dimensional crosstabs

One-dimensional crosstabs show a summary row (or column) for the categories of a single dimension. For example, if Payment is the chosen column dimension and Amount Paid is the summary category, the crosstab in Figure 22.2 shows the amount paid using each method.

**Figure 22.2** One-dimensional crosstab



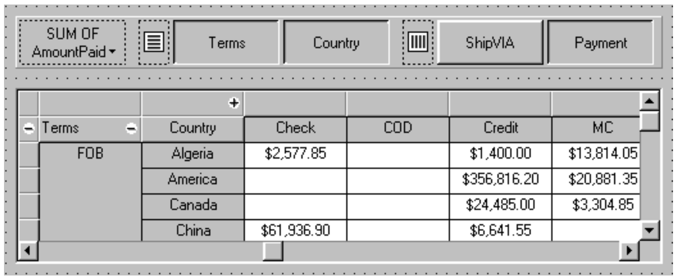
| Payment | | | | | |
|---|---|---|---|---|---|
| AmEx | Cash | Check | COD | Credit | MC |
| $134,753.40 | $164,003.65 | $270,492.15 | $33,776.55 | $1,332,430.25 | $250,163.25 |

## Multidimensional crosstabs

Multidimensional crosstabs use additional dimensions for the rows and/or columns. For example, a two-dimensional crosstab could show amounts paid by payment method for each country.

A three-dimensional crosstab could show amounts paid by payment method and terms by country, as shown in Figure 22.3.

**Figure 22.3** Three-dimensional crosstab



| Terms | Country | Check | COD | Credit | MC |
|---|---|---|---|---|---|
| FOB | Algeria | $2,577.85 | | $1,400.00 | $13,814.05 |
| | America | | | $356,816.20 | $20,881.35 |
| | Canada | | | $24,485.00 | $3,304.85 |
| | China | $61,936.90 | | $6,641.55 | |

## Guidelines for using decision support components

The decision support components listed on page 22-1 can be used together to present multidimensional data as tables and graphs. More than one grid or graph can be attached to each dataset. More than one instance of *TDecisionPivot* can be used to display the data from different perspectives at runtime.

To create a form with tables and graphs of multidimensional data, follow these steps:

1. Create a form.
2. Add these components to the form and use the Object Inspector to bind them as indicated:
   - A dataset, usually *TDecisionQuery* (for details, see "Creating decision datasets with the Decision Query editor" on page 22-6) or *TQuery*
   - A decision cube, *TDecisionCube*, bound to the dataset by setting its *DataSet* property to the dataset's name
   - A decision source, *TDecisionSource*, bound to the decision cube by setting its *DecisionCube* property to the decision cube's name
3. Add a decision pivot, *TDecisionPivot*, and bind it to the decision source with the Object Inspector by setting its *DecisionSource* property to the appropriate decision source name. The decision pivot is optional but useful; it lets the form developer and end users change the dimensions displayed in decision grids or decision graphs by pushing buttons.

   In its default orientation, horizontal, buttons on the left side of the decision pivot apply to fields on the left side of the decision grid (rows); buttons on the right side apply to fields at the top of the decision grid (columns).

   You can determine where the decision pivot's buttons appear by setting its *GroupLayout* property to *xtVertical*, *xtLeftTop*, or *xtHorizontal* (the default). For more information on decision pivot properties, see "Using decision pivots" on page 22-10.
4. Add one or more decision grids and graphs, bound to the decision source. For details, see "Creating and using decision grids" on page 22-11 and "Creating and using decision graphs" on page 22-13.
5. Use the Decision Query editor or *SQL* property of *TDecisionQuery* (or *TQuery*) to specify the tables, fields, and summaries to display in the grid or graph. The last field of the SQL SELECT should be the summary field. The other fields in the SELECT must be GROUP BY fields. For instructions, see "Creating decision datasets with the Decision Query editor" on page 22-6.
6. Set the *Active* property of the decision query (or alternate dataset component) to *True*.

**7** Use the decision grid and graph to show and chart different data dimensions. See "Using decision grids" on page 22-11 and "Using decision graphs" on page 22-14 for instructions and suggestions.

For an illustration of all decision support components on a form, see Figure 22.1 on page 22-2.

## Using datasets with decision support components

The only decision support component that binds directly to a dataset is the decision cube, *TDecisionCube. TDecisionCube* expects to receive data with groups and summaries defined by an SQL statement of an acceptable format. The GROUP BY phrase must contain the same non-summarized fields (and in the same order) as the SELECT phrase, and summary fields must be identified.

The decision query component, *TDecisionQuery,* is a specialized form of *TQuery*. You can use TDecisionQuery to more simply define the setup of dimensions (rows and columns) and summary values used to supply data to decision cubes (*TDecisionCube).* You can also use an ordinary *TQuery* or other BDE-enabled dataset as a dataset for *TDecisionCube*, but the correct setup of the dataset and *TDecisionCube* are then the responsibility of the designer.

To work correctly with the decision cube, all projected fields in the dataset must either be dimensions or summaries. The summaries should be additive values (like sum or count), and should represent totals for each combination of dimension values. For maximum ease of setup, sums should be named "Sum..." in the dataset while counts should be named "Count...".

The Decision Cube can pivot, subtotal, and drill-in correctly only for summaries whose cells are additive. (SUM and COUNT are additive, while AVERAGE, MAX, and MIN are not.) Build pivoting crosstab displays only for grids that contain only additive aggregators. If you are using non-additive aggregators, use a static decision grid that does not pivot, drill, or subtotal.

Since averages can be calculated using SUM divided by COUNT, a pivoting average is added automatically when SUM and COUNT dimensions for a field are included in the dataset. Use this type of average in preference to an average calculated using an AVERAGE statement.

Averages can also be calculated using COUNT(*). To use COUNT(*) to calculate averages, include a "COUNT(*) COUNTALL" selector in the query. If you use COUNT(*) to calculate averages, the single aggregator can be used for all fields. Use COUNT(*) only in cases where none of the fields being summarized include blank values, or where a COUNT aggregator is not available for every field.

## Creating decision datasets with TQuery or TTable

If you use an ordinary *TQuery* component as a decision dataset, you must manually set up the SQL statement, taking care to supply a GROUP BY phrase which contains the same fields (and in the same order) as the SELECT phrase.

The SQL should look similar to this:

```
SELECT ORDERS."Terms", ORDERS."ShipVIA",
  ORDERS."PaymentMethod", SUM( ORDERS."AmountPaid" )
FROM "ORDERS.DB" ORDERS
GROUP BY ORDERS."Terms", ORDERS."ShipVIA", ORDERS."PaymentMethod"
```

The ordering of the SELECT fields should match the ordering of the GROUP BY fields.

With *TTable*, you must supply information to the decision cube about which of the fields in the query are grouping fields, and which are summaries. To do this, Fill in the Dimension Type for each field in the *DimensionMap* of the Decision Cube. You must indicate whether each field is a dimension or a summary, and if a summary, you must provide the summary type. Since pivoting averages depend on SUM/COUNT calculations, you must also provide the base field name to allow the decision cube to match pairs of SUM and COUNT aggregators.

## Creating decision datasets with the Decision Query editor

All data used by the decision support components passes through the decision cube, which accepts a specially formatted dataset most easily produced by an SQL query. See "Using datasets with decision support components" on page 22-5 for more information.

While both *TTable* and *TQuery* can be used as decision datasets, it is easier to use *TDecisionQuery*; the Decision Query editor supplied with it can be used to specify tables, fields, and summaries to appear in the decision cube and will help you set up the SELECT and GROUP BY portions of the SQL correctly.

To use the Decision Query editor:

1 Select the decision query component on the form, then right-click and choose Decision Query editor. The Decision Query editor dialog box appears.

2 Choose the database to use.

3 For single-table queries, click the Select Table button.

   For more complex queries involving multi-table joins, click the Query Builder button to display the SQL Builder or type the SQL statement into the edit box on the SQL tab page.

4 Return to the Decision Query editor dialog box.

5 In the Decision Query editor dialog box, select fields in the Available Fields list box and assign them to be either Dimensions or Summaries by clicking the appropriate right arrow button. As you add fields to the Summaries list, select from the menu displayed the type of summary to use: sum, count, or average.

6 By default, all fields and summaries defined in the *SQL* property of the decision query appear in the Active Dimensions and Active Summaries list boxes. To remove a dimension or summary, select it in the list and click the left arrow beside the list, or double-click the item to remove. To add it back, select it in the Available Fields list box and click the appropriate right arrow.

Once you define the contents of the decision cube, you can further manipulate dimension display with its *DimensionMap* property and the buttons of *TDecisionPivot*. For more information, see the next section, "Using decision cubes," "Using decision sources" on page 22-9, and "Using decision pivots" on page 22-10.

**Note** When you use the Decision Query editor, the query is initially handled in ANSI-92 SQL syntax, then translated (if necessary) into the dialect used by the server. The Decision Query editor reads and displays only ANSI standard SQL. The dialect translation is automatically assigned to the *TDecisionQuery*'s SQL property. To modify a query, edit the ANSI-92 version in the Decision Query rather then the SQL property.

## Using decision cubes

The decision cube component, *TDecisionCube*, is a multidimensional data store that fetches its data from a dataset (typically a specially structured SQL statement entered through *TDecisionQuery* or *TQuery*). The data is stored in a form that makes its easy to pivot (that is, change the way in which the data is organized and summarized) without needing to run the query a second time.

### Decision cube properties and events

The *DimensionMap* properties of *TDecisionCube* not only control which dimensions and summaries appear but also let you set date ranges and specify the maximum number of dimensions the decision cube may support. You can also indicate whether or not to display data during design. You can display names, (categories) values, subtotals, or data. Display of data at design time can be time consuming, depending on the data source.

When you click the ellipsis next to *DimensionMap* in the Object Inspector, the Decision Cube editor dialog box appears. You can use its pages and controls to set the *DimensionMap* properties.

The *OnRefresh* event fires whenever the decision cube cache is rebuilt. Developers can access the new dimension map and change it at that time to free up memory, change the maximum summaries or dimensions, and so on. *OnRefresh* is also useful if users access the Decision Cube editor; application code can respond to user changes at that time.

## Using the Decision Cube editor

You can use the Decision Cube editor to set the *DimensionMap* properties of decision cubes. You can display the Decision Cube editor through the Object Inspector, as described in the previous section, or by right-clicking a decision cube on a form at design time and choosing Decision Cube editor.

The Decision Cube Editor dialog box has two tabs:

- Dimension Settings, used to activate or disable available dimensions, rename and reformat dimensions, put dimensions in a permanently drilled state, and set date ranges to display.
- Memory Control, used to set the maximum number of dimensions and summaries that can be active at one time, to display information about memory usage, and to determine the names and data that appear at design time.

### Viewing and changing dimension settings

To view the dimension settings, display the Decision Cube editor and click the Dimension Settings tab. Then, select a dimension or summary in the Available Fields list. Its information appears in the boxes on the right side of the editor:

- To change the dimension or summary name that appears in the decision pivot, decision grid, or decision graph, enter a new name in the Display Name edit box.
- To determine whether the selected field is a dimension or summary, read the text in the Type edit box. If the dataset is a *TTable* component, you can use Type to specify whether the selected field is a dimension or summary.
- To disable or activate the selected dimension or summary, change the setting in the Active Type drop-down list box: Active, As Needed, or Inactive. Disabling a dimension or setting it to As Needed saves memory.
- To change the format of that dimension or summary, enter a format string in the Format edit box.
- To display that dimension or summary by Year, Quarter, or Month, change the setting in the Binning drop-down list box. Note that you can choose Set in the Binning list box to put the selected dimension or summary in a permanently "drilled down" state. This can be useful for saving memory when a dimension has many values. For more information, see "Decision support components and memory control" on page 22-20.
- To determine the starting value for ranges, or the drill-down value for a "Set" dimension, first choose the appropriate Grouping value in the Grouping drop-down, and then enter the starting range value or permanent drill-down value in the Initial Value drop-down list.

### Setting the maximum available dimensions and summaries

To determine the maximum number of dimensions and summaries available for decision pivots, decision grids, and decision graphs bound to the selected decision cube, display the Decision Cube editor and click the Memory Control tab. Use the edit controls to adjust the current settings, if necessary. These settings help to control the amount of memory required by the decision cube. For more information, see "Decision support components and memory control" on page 22-20.

### Viewing and changing design options

To determine how much information appears at design time, display the Decision Cube editor and click the Memory Control tab. Then, check the setting that indicates which names and data to display. Display of data or field names at design time can cause performance delays in some cases because of the time needed to fetch the data.

## Using decision sources

The decision source component, *TDecisionSource,* defines the current pivot state of decision grids or decision graphs. Any two objects which use the same decision source also share pivot states.

### Properties and events

The following are some special properties and events that control the appearance and behavior of decision sources:

- The *ControlType* property of *TDecisionSource* indicates whether the decision pivot buttons should act like check boxes (multiple selections) or radio buttons (mutually exclusive selections).
- The *SparseCols* and *SparseRows* properties of *TDecisionSource* indicate whether to display columns or rows with no values; if *True*, sparse columns or rows are displayed.
- *TDecisionSource* has the following events:
  - *OnLayoutChange* occurs when the user performs pivots or drill-downs that reorganize the data.
  - *OnNewDimensions* occurs when the data is completely altered, such as when the summary or dimension fields are altered.
  - *OnSummaryChange* occurs when the current summary is changed.
  - *OnStateChange* occurs when the Decision Cube activates or deactivates.

- *OnBeforePivot* occurs when changes are committed but not yet reflected in the user interface. Developers have an opportunity to make changes, for example, in capacity or pivot state, before application users see the result of their previous action.
- *OnAfterPivot* fires after a change in pivot state. Developers can capture information at that time.

## Using decision pivots

The decision pivot component, *TDecisionPivot,* lets you open or close decision cube dimensions, or fields, by pressing buttons. When a row or column is opened by pressing a *TDecisionPivot* button, the corresponding dimension appears on the *TDecisionGrid* or *TDecisionGraph* component. When a dimension is closed, its detailed data doesn't appear; it collapses into the totals of other dimensions. A dimension may also be in a "drilled" state, where only the summaries for a particular value of the dimension field appear.

You can also use the decision pivot to reorganize dimensions displayed in the decision grid and decision graph. Just drag a button to the row or column area or reorder buttons within the same area.

For illustrations of decision pivots at design time, see Figures 22.1, 22.2, and 22.3.

### Decision pivot properties

The following are some special properties that control the appearance and behavior of decision pivots:

- The first properties listed for *TDecisionPivot* define its overall behavior and appearance. You might want to set *ButtonAutoSize* to *False* for *TDecisionPivot* to keep buttons from expanding and contracting as you adjust the size of the component.
- The *Groups* property of *TDecisionPivot* defines which dimension buttons appear. You can display the row, column, and summary selection button groups in any combination. Note that if you want more flexibility over the placement of these groups, you can place one *TDecisionPivot* on your form which contains only rows in one location, and a second which contains only columns in another location.
- Typically, *TDecisionPivot* is added above *TDecisionGrid*. In its default orientation, horizontal, buttons on the left side of *TDecisionPivot* apply to fields on the left side of *TDecisionGrid* (rows); buttons on the right side apply to fields at the top of *TDecisionGrid* (columns).
- You can determine where *TDecisionPivot*'s buttons appear by setting its *GroupLayout* property to *xtVertical*, *xtLeftTop*, or *xtHorizontal* (the default, described in the previous paragraph).

## Creating and using decision grids

Decision grid components, *TDecisionGrid*, present cross-tabulated data in table form. These tables are also called crosstabs, described on page 22-2. Figure 22.1 on page 22-2 shows a decision grid on a form at design time.

### Creating decision grids

To create a form with one or more tables of cross-tabulated data,

1. Follow steps 1–3 listed under “Guidelines for using decision support components” on page 22-4.
2. Add one or more decision grid components (*TDecisionGrid*) and bind them to the decision source, *TDecisionSource*, with the Object Inspector by setting their *DecisionSource* property to the appropriate decision source component.
3. Continue with steps 5–7 listed under “Guidelines for using decision support components.”

For a description of what appears in the decision grid and how to use it, see “Using decision grids” on page 22-11.

To add a graph to the form, follow the instructions in “Creating decision graphs” on page 22-13.

### Using decision grids

The decision grid component, *TDecisionGrid*, displays data from decision cubes (*TDecisionCube*) bound to decision sources (*TDecisionSource*).

By default, the grid appears with dimension fields at its left side and/or top, depending on the grouping instructions defined in the dataset. Categories, one for each data value, appear under each field. You can

- Open and close dimensions
- Reorganize, or pivot, rows and columns
- Drill down for detail
- Limit dimension selection to a single dimension for each axis

For more information about special properties and events of the decision grid, see “Decision grid properties” on page 22-12.

#### Opening and closing decision grid fields

If a plus sign (+) appears in a dimension or summary field, one or more fields to its right are closed (hidden). You can open additional fields and categories by clicking the sign. A minus sign (-) indicates a fully opened (expanded) field. When you click the sign, the field closes. This outlining feature can be disabled; see “Decision grid properties” on page 22-12 for details.

### Reorganizing rows and columns in decision grids

You can drag row and column headings to new locations within the same axis or to the other axis. In this way, you can reorganize the grid and see the data from new perspectives as the data groupings change. This pivoting feature can be disabled; see "Decision grid properties" on page 22-12 for details.

If you included a decision pivot, you can push and drag its buttons to reorganize the display. See "Using decision pivots" on page 22-10 for instructions.

### Drilling down for detail in decision grids

You can drill down to see more detail in a dimension.

For example, if you right-click a category label (row heading) for a dimension with others collapsed beneath it, you can choose to drill down and only see data for that category. When a dimension is drilled, you do not see the category labels for that dimension displayed on the grid, since only the records for a single category value are being displayed. If you have a decision pivot on the form, it displays category values and lets you change to other values if you want.

To drill down into a dimension,

- Right-click a category label and choose Drill In To This Value, or
- Right-click a pivot button and choose Drilled In.

To make the complete dimension active again,

- Right-click the corresponding pivot button, or
- right-click the decision grid in the upper-left corner and select the dimension.

### Limiting dimension selection in decision grids

You can change the *ControlType* property of the decision source to determine whether more than one dimension can be selected for each axis of the grid. For more information, see "Using decision sources" on page 22-9.

## Decision grid properties

The decision grid component, *TDecisionGrid*, displays data from the *TDecisionCube* component bound to *TDecisionSource*. By default, data appears in a grid with category fields on the left side and top of the grid.

The following are some special properties that control the appearance and behavior of decision grids:

- *TDecisionGrid* has unique properties for each dimension. To set these, choose *Dimensions* in the Object Inspector, then select a dimension. Its properties then appear in the Object Inspector: *Alignment* defines the alignment of category labels for that dimension, *Caption* can be used to override the default dimension name, *Color* defines the color of category labels, *FieldName* displays the name of the active dimension, *Format* can hold any standard format for that data type, and *Subtotals* indicates whether to display subtotals for that dimension. With summary fields, these same properties are used to changed the appearance of the data that appears in the summary area of the grid. When you're through setting dimension properties, either click a component in the form or choose a component in the drop-down list box at the top of the Object Inspector.

- The *Options* property of *TDecisionGrid* lets you control display of grid lines (*cgGridLines* = *True*), enabling of outline features (collapse and expansion of dimensions with + and - indicators; *cgOutliner* = *True*), and enabling of drag-and-drop pivoting (*cgPivotable* = *True*).
- The *OnDecisionDrawCell* event of *TDecisionGrid* gives you a chance to change the appearance of each cell as it is drawn. The event passes the *String*, *Font*, and *Color* of the current cell as reference parameters. You are free to alter those parameters to achieve effects such as special colors for negative values. In addition to the *DrawState* which is passed by *TCustomGrid*, the event passes *TDecisionDrawState*, which can be used to determine what type of cell is being drawn. Further information about the cell can be fetched using the *Cells*, *CellValueArray*, or *CellDrawState* functions.
- The *OnDecisionExamineCell* event of *TDecisionGrid* lets you hook the right-click-on-event to data cells, and is intended to allow a program to display information (such as detail records) about that particular data cell. When the user right-clicks a data cell, the event is supplied with all the information which is was used to compose the data value, including the currently active summary value and a *ValueArray* of all the dimension values which were used to create the summary value.

## Creating and using decision graphs

Decision graph components, *TDecisionGraph,* present cross-tabulated data in graphic form. Each decision graph shows the value of a single summary, such as Sum, Count, or Avg, charted for one or more dimensions. For more information on crosstabs, see page 22-3. For illustrations of decision graphs at design time, see Figure 22.1 on page 22-2 and Figure 22.4 on page 22-15.

### Creating decision graphs

To create a form with one or more decision graphs,

1. Follow steps 1–3 listed under "Guidelines for using decision support components" on page 22-4.
2. Add one or more decision graph components (*TDecisionGraph*) and bind them to the decision source, *TDecisionSource,* with the Object Inspector by setting their *DecisionSource* property to the appropriate decision source component.

3 Continue with steps 5–7 listed under "Guidelines for using decision support components."

4 Finally, right-click the graph and choose Edit Chart to modify the appearance of the graph series. You can set template properties for each graph dimension, then set individual series properties to override these defaults. For details, see "Customizing decision graphs" on page 22-16.

For a description of what appears in the decision graph and how to use it, see the next section, "Using decision graphs."

To add a decision grid—or crosstab table—to the form, follow the instructions in "Creating and using decision grids" on page 22-11.

## Using decision graphs

The decision graph component, *TDecisionGraph*, displays fields from the decision source (*TDecisionSource*) as a dynamic graph that changes when data dimensions are opened, closed, dragged and dropped, or rearranged with the decision pivot (*TDecisionPivot*).

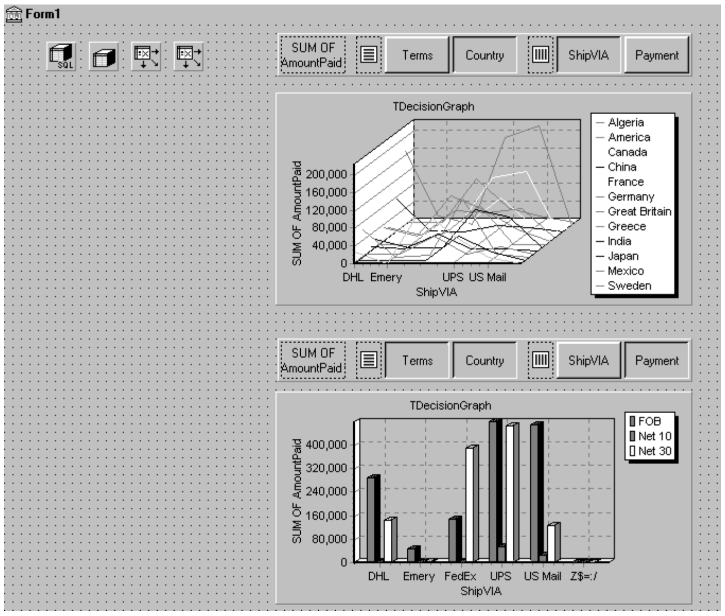
Graphed data comes from a specially formatted dataset such as *TDecisionQuery*. For an overview of how the decision support components handle and arrange this data, see page 22-1.

By default, the first row dimension appears as the x-axis and the first column dimension appears as the y-axis.

You can use decision graphs instead of or in addition to decision grids, which present cross-tabulated data in tabular form. Decision grids and decision graphs that are bound to the same decision source present the same data dimensions. To show different summary data for the same dimensions, you can bind more than one decision graph to the same decision source. To show different dimensions, bind decision graphs to different decision sources.

For example, in Figure 22.4 the first decision pivot and graph are bound to the first decision source and the second decision pivot and graph are bound to the second. So, each graph can show different dimensions.

**Figure 22.4** Decision graphs bound to different decision sources



For more information about what appears in a decision graph, see the next section, "The decision graph display."

To create a decision graph, see the previous section, "Creating decision graphs."

For a discussion of decision graph properties and how to change the appearance and behavior of decision graphs, see "Customizing decision graphs" on page 22-16.

## The decision graph display

By default, the decision graph plots summary values for categories in the first active row field (along the y-axis) against values in the first active column field (along the x-axis). Each graphed category appears as a separate series.

If only one dimension is selected—for example, by clicking only one *TDecisionPivot* button—only one series is graphed.

If you used a decision pivot, you can push its buttons to determine which decision cube fields (dimensions) are graphed. To exchange graph axes, drag the decision pivot dimension buttons from one side of the separator space to the other. If you have a one-dimensional graph with all buttons on one side of the separator space, you can use the Row or Column icon as a drop target for adding buttons to the other side of the separator and making the graph multidimensional.

If you only want one column and one row to be active at a time, you can set the *ControlType* property for *TDecisionSource* to *xtRadio*. Then, there can be only one active field at a time for each decision cube axis, and the decision pivot's functionality will correspond to the graph's behavior. *xtRadioEx* works the same as *xtRadio*, but does not allow the state where all row or all columns dimensions are closed.

When you have both a decision grid and graph connected to the same *TDecisionSource*, you'll probably want to set *ControlType* back to *xtCheck* to correspond to the more flexible behavior of *TDecisionGrid*.

## Customizing decision graphs

The decision graph component, *TDecisionGraph*, displays fields from the decision source (*TDecisionSource*) as a dynamic graph that changes when data dimensions are opened, closed, dragged and dropped, or rearranged with the decision pivot (*TDecisionPivot*). You can change the type, colors, marker types for line graphs, and many other properties of decision graphs.

To customize a graph,

1. Right-click it and choose Edit Chart. The Chart Editing dialog box appears.
2. Use the Chart page of the Chart Editing dialog box to view a list of visible series, select the series definition to use when two or more are available for the same series, change graph types for a template or series, and set overall graph properties.

   The Series list on the Chart page shows all decision cube dimensions (preceded by Template:) and currently visible categories. Each category, or series, is a separate object. You can:

   - Add or delete series derived from existing decision-graph series. Derived series can provide annotations for existing series or represent values calculated from other series.
   - Change the default graph type, and change the title of templates and series.

   For a description of the other Chart page tabs, search for the following topic in online Help: "Chart page (Chart Editing dialog box)."
3. Use the Series page to establish dimension templates, then customize properties for each individual graph series.

By default, all series are graphed as bar graphs and up to 16 default colors are assigned. You can edit the template type and properties to create a new default. Then, as you pivot the decision source to different states, the template is used to dynamically create the series for each new state. For template details, see "Setting decision graph template defaults" on page 22-17.

To customize individual series, follow the instructions in "Customizing decision graph series" on page 22-18.

For a description of each Series page tab, search for the following topic in online Help: "Series page (Chart Editing dialog box)."

## Setting decision graph template defaults

Decision graphs display the values from two dimensions of the decision cube: one dimension is displayed as an axis of the graph, and the other is used to create a set of series. The template for that dimension provides default properties for those series (such as whether the series are bar, line, area, and so on). As users pivot from one state to another, any required series for the dimension are created using the series type and other defaults specified in the template.

A separate template is provided for cases where users pivot to a state where only one dimension is active. A one-dimensional state is often represented with a pie chart, so a separate template is provided for this case.

You can

- Change the default graph type.
- Change other graph template properties.
- View and set overall graph properties.

### Changing the default decision graph type

To change the default graph type,

1 Select a template in the Series list on the Chart page of the Chart Editing dialog box.

2 Click the Change button.

3 Select a new type and close the Gallery dialog box.

### Changing other decision graph template properties

To change color or other properties of a template,

1 Select the Series page at the top of the Chart Editing dialog box.

2 Choose a template in the drop-down list at the top of the page.

3 Choose the appropriate property tab and select settings.

### Viewing overall decision graph properties

To view and set decision graph properties other than type and series,

1 Select the Chart page at the top of the Chart Editing dialog box.

2 Choose the appropriate property tab and select settings.

## Customizing decision graph series

The templates supply many defaults for each decision cube dimension, such as graph type and how series are displayed. Other defaults, such as series color, are defined by *TDecisionGraph*. If you want you can override the defaults for each series.

The templates are intended for use when you want the program to create the series for categories as they are needed, and discard them when they are no longer needed. If you want, you can set up custom series for specific category values. To do this, pivot the graph so its current display has a series for the category you want to customize. When the series is displayed on the graph, you can use the Chart editor to

- Change the graph type.
- Change other series properties.
- Save specific graph series that you have customized.

To define series templates and set overall graph defaults, see "Setting decision graph template defaults" on page 22-17.

### Changing the series graph type

By default, each series has the same graph type, defined by the template for its dimension. To change all series to the same graph type, you can change the template type. See "Changing the default decision graph type" on page 22-17 for instructions.

To change the graph type for a single series,

1 Select a series in the Series list on the Chart page of the Chart editor.

2 Click the Change button.

3 Select a new type and close the Gallery dialog box.

4 Check the Save Series check box.

### Changing other decision graph series properties

To change color or other properties of a decision graph series,

1 Select the Series page at the top of the Chart Editing dialog box.

2 Choose a series in the drop-down list at the top of the page.

3 Choose the appropriate property tab and select settings.

4 Check the Save Series check box.

### Saving decision graph series settings

By default, only settings for templates are saved at design time. Changes made to specific series are only saved if the Save box is checked for that series in the Chart Editing dialog box.

Saving series can be memory intensive, so if you don't need to save them you can uncheck the Save box.

## Decision support components at runtime

At runtime, users can perform many operations by left-clicking, right-clicking, and dragging visible decision support components. These operations, discussed earlier in this chapter, are summarized below.

### Decision pivots at runtime

Users can:

- Left-click the summary button at the left end of the decision pivot to display a list of available summaries. They can use this list to change the summary data displayed in decision grids and decision graphs.
- Right-click a dimension button and choose to:
  - Move it from the row area to the column area or the reverse.
  - Drill In to display detail data.
- Left-click a dimension button following the Drill In command and choose:
  - Open Dimension to move back to the top level of that dimension.
  - All Values to toggle between displaying just summaries and summaries plus all other values in decision grids.
  - From a list of available categories for that dimension, a category to drill into for detail values.
- Left-click a dimension button to open or close that dimension.
- Drag and drop dimension buttons from the row area to the column area and the reverse; they can drop them next to existing buttons in that area or onto the row or column icon.

### Decision grids at runtime

Users can:

- Right-click within the decision grid and choose to:
  - Toggle subtotals on and off for individual data groups, for all values of a dimension, or for the whole grid.
  - Display the Decision Cube editor, described on page 22-8.
  - Toggle dimensions and summaries open and closed.
- Click + and – within the row and column headings to open and close dimensions.
- Drag and drop dimensions from rows to columns and the reverse.

### Decision graphs at runtime

Users can drag from side to side or up and down in the graph grid area to scroll through off-screen categories and values.

## Decision support components and memory control

When a dimension or summary is loaded into the decision cube, it takes up memory. Adding a new summary increases memory consumption linearly: that is, a decision cube with two summaries uses twice as much memory as the same cube with only one summary, a decision cube with three summaries uses three times as much memory as the same cube with one summary, and so on. Memory consumption for dimensions increases more quickly. Adding a dimension with 10 values increases memory consumption by a factor of 10. Adding a dimension with 100 values increases memory consumption 100 times. Thus adding dimensions to a decision cube can have a dramatic effect on memory use, and can quickly lead to performance problems. This effect is especially pronounced when adding dimensions that have many values.

The decision support components have a number of settings to help you control how and when memory is used. For more information on the properties and techniques mentioned here, look up *TDecisionCube* in the online Help.

### Setting maximum dimensions, summaries, and cells

The decision cube's *MaxDimensions* and *MaxSummaries* properties can be used with the *CubeDim.ActiveFlag* property to control how many dimensions and summaries can be loaded at a time. You can set the maximum values on the Cube Capacity page of the Decision Cube editor to place some overall control on how many dimensions or summaries can be brought into memory at the same time.

Limiting the number of dimensions or summaries provides a rough limit on the amount of memory used by the decision cube. However, it does not distinguish between dimensions with many values and those with only a few. For greater control of the absolute memory demands of the decision cube, you can also limit the number of cells in the cube. Set the maximum number of cells on the Cube Capacity page of the Decision Cube editor.

## Setting dimension state

The *ActiveFlag* property controls which dimensions get loaded. You can set this property on the Dimension Settings tab of the Decision Cube editor using the Activity Type control. When this control is set to *Active*, the dimension is loaded unconditionally, and will always take up space. Note that the number of dimensions in this state must always be less than *MaxDimensions*, and the number of summaries set to *Active* must be less than *MaxSummaries*. You should set a dimension or summary to *Active* only when it is critical that it be available at all times. An *Active* setting decreases the ability of the cube to manage the available memory.

When *ActiveFlag* is set to *AsNeeded*, a dimension or summary is loaded only if it can be loaded without exceeding the *MaxDimensions*, *MaxSummaries,* or *MaxCells* limit. The decision cube will swap dimensions and summaries that are marked *AsNeeded* in and out of memory to keep within the limits imposed by *MaxCells*, *MaxDimensions*, and *MaxSummaries*. Thus, a dimension or summary may not be loaded in memory if it is not currently being used. Setting dimensions that are not used frequently to *AsNeeded* results in better loading and pivoting performance, although there will be a time delay to access dimensions which are not currently loaded.

## Using paged dimensions

When Binning is set to Set on the Dimension Settings tab of the Decision cube editor and Start Value is not NULL, the dimension is said to be “paged,” or “permanently drilled down.” You can access data for just a single value of that dimension at a time, although you can programmatically access a series of values sequentially. Such a dimension may not be pivoted or opened.

It is extremely memory intensive to include dimensional data for dimensions that have very large numbers of values. By making such dimensions paged, you can display summary information for one value at a time. Information is usually easier to read when displayed this way, and memory consumption is much easier to manage.

Chapter

23

# Connecting to databases

Most dataset components can connect directly to a database server. Once connected, the dataset communicates with the server automatically. When you open the dataset, it populates itself with data from the server, and when you post records, they are sent back the server and applied. A single connection component can be shared by multiple datasets, or each dataset can use its own connection.

Each type of dataset connects to the database server using its own type of connection component, which is designed to work with a single data access mechanism. The following table lists these data access mechanisms and the associated connection components:

**Table 23.1** Database connection components

| Data access mechanism | Connection component |
|---|---|
| Borland Database Engine (BDE) | TDatabase |
| ActiveX Data Objects (ADO) | TADOConnection |
| dbExpress | TSQLConnection |
| InterBase Express | TIBDatabase |

**Note** For a discussion of some pros and cons of each of these mechanisms, see “Using databases” on page 19-1.

The connection component provides all the information necessary to establish a database connection. This information is different for each type of connection component:

- For information about describing a BDE-based connection, see “Identifying the database” on page 26-14.
- For information about describing an ADO-based connection, see “Connecting to a data store using TADOConnection” on page 27-3.

- For information about describing a dbExpress connection, see "Setting up TSQLConnection" on page 28-3.
- For information about describing an InterBase Express connection, see the online help for *TIBDatabase*.

Although each type of dataset uses a different connection component, they are all descendants of *TCustomConnection*. They all perform many of the same tasks and surface many of the same properties, methods, and events. This chapter discusses many of these common tasks.

## Using implicit connections

No matter what data access mechanism you are using, you can always create the connection component explicitly and use it to manage the connection to and communication with a database server. For BDE-enabled and ADO-based datasets, you also have the option of describing the database connection through properties of the dataset and letting the dataset generate an implicit connection. For BDE-enabled datasets, you specify an implicit connection using the *DatabaseName* property. For ADO-based datasets, you use the *ConnectionString* property.

When using an implicit connection, you do not need to explicitly create a connection component. This can simplify your application development, and the default connection you specify can cover a wide variety of situations. For complex, mission-critical client/server applications with many users and different requirements for database connections, however, you should create your own connection components to tune each database connection to your application's needs. Explicit connection components give you greater control. For example, you need to access the connection component to perform the following tasks:

- Customize database server login support. (Implicit connections display a default login dialog to prompt the user for a user name and password.)
- Control transactions and specify transaction isolation levels.
- Execute SQL commands on the server without using a dataset.
- Perform actions on all open datasets that are connected to the same database.

In addition, if you have multiple datasets that all use the same server, it can be easier to use an connection component, so that you only have to specify the server to use in one place. That way, if you later change the server, you do not need to update several dataset components: only the connection component.

## Controlling connections

Before you can establish a connection to a database server, your application must provide certain key pieces of information that describe the desired server. Each type of connection component surfaces a different set of properties to let you identify the server. In general, however, they all provide a way for you to name the server you want and supply a set of connection parameters that control how the connection is formed. Connection parameters vary from server to server. They can include information such as user name and password, the maximum size of BLOB fields, SQL roles, and so on.

Once you have identified the desired server and any connection parameters, you can use the connection component to explicitly open or close a connection. The connection component generates events when it opens or closes a connection that you can use to customize the response of your application to changes in the database connection.

### Connecting to a database server

There are two ways to connect to a database server using a connection component:

- Call the *Open* method.
- Set the *Connected* property to *True*.

Calling the *Open* method sets *Connected* to *True*.

**Note** When a connection component is not connected to a server and an application attempts to open one of its associated datasets, the dataset automatically calls the connection component's *Open* method.

When you set *Connected* to *True*, the connection component first generates a *BeforeConnect* event, where you can perform any initialization. For example, you can use this event to alter connection parameters.

After the *BeforeConnect* event, the connection component may display a default login dialog, depending on how you choose to control server login. It then passes the user name and password to the driver, opening a connection.

Once the connection is open, the connection component generates an *AfterConnect* event, where you can perform any tasks that require an open connection.

**Note** Some connection components generate additional events as well when establishing a connection.

Once a connection is established, it is maintained as long as there is at least one active dataset using it. When there are no more active datasets, the connection component drops the connection. Some connection components surface a *KeepConnection* property that allows the connection to remain open even if all the datasets that use it are closed. If *KeepConnection* is *True*, the connection is maintained. For connections to remote database servers, or for applications that frequently open and close datasets, setting *KeepConnection* to *True* reduces network traffic and speeds up the application. If *KeepConnection* is *False,* the connection is dropped when there are no active datasets using the database. If a dataset that uses the database is later opened, the connection must be reestablished and initialized.

### Disconnecting from a database server

There are two ways to disconnect a server using a connection component:

- Set the *Connected* property to *False*.
- Call the *Close* method.

Calling *Close* sets *Connected* to *False*.

When *Connected* is set to *False*, the connection component generates a *BeforeDisconnect* event, where you can perform any cleanup before the connection closes. For example, you can use this event to cache information about all open datasets before they are closed.

After the *BeforeConnect* event, the connection component closes all open datasets and disconnects from the server.

Finally, the connection component generates an *AfterDisconnect* event, where you can respond to the change in connection status, such as enabling a Connect button in your user interface.

**Note** Calling *Close* or setting *Connected* to *False* disconnects from a database server even if the connection component has a *KeepConnection* property that is *True*.

## Controlling server login

Most remote database servers include security features to prohibit unauthorized access. Usually, the server requires a user name and password login before permitting database access.

At design time, if a server requires a login, a standard login dialog box prompts for a user name and password when you first attempt to connect to the database.

At runtime, there are three ways you can handle a server's request for a login:

- Let the default login dialog and processes handle the login. This is the default approach. Set the *LoginPrompt* property of the connection component to *True* (the default) and add DBLogDlg to the uses clause of the unit that declares the connection component. Your application displays the standard login dialog box when the server requests a user name and password.

- Supply the login information before the login attempt. Each type of connection component uses a different mechanism for specifying the user name and password:
  - For BDE, dbExpress, and InterBase express datasets, the user name and password connection parameters can be accessed through the *Params* property. (For BDE datasets, the parameter values can also be associated with a BDE alias, while for dbExpress datasets, they can also be associated with a connection name).
  - For ADO datasets, the user name and password can be included in the *ConnectionString* property (or provided as parameters to the *Open* method).

  If you specify the user name and password before the server requests them, be sure to set the *LoginPrompt* to *False*, so that the default login dialog does not appear. For example, the following code sets the user name and password on a SQL connection component in the *BeforeConnect* event handler, decrypting an encrypted password that is associated with the current connection name:

  ```
  procedure TForm1.SQLConnectionBeforeConnect(Sender: TObject);
  begin
    with Sender as TSQLConnection do
    begin
      if LoginPrompt = False then
      begin
        Params.Values['User_Name'] := 'SYSDBA';
        Params.Values['Password'] := Decrypt(Params.Values['Password']);
      end;
    end;
  end;
  ```

  Note that setting the user name and password at design-time or using hard-coded strings in code causes the values to be embedded in the application's executable file. This still leaves them easy to find, compromising server security.

- Provide your own custom handling for the login event. The connection component generates an event when it needs the user name and password.
  - For *TDatabase*, *TSQLConnection*, and *TIBDatabase*, this is an *OnLogin* event. The event handler has two parameters, the connection component, and a local copy of the user name and password parameters in a string list. (*TSQLConnection* includes the database parameter as well). You must set the *LoginPrompt* property to *True* for this event to occur. Having a *LoginPrompt* value of *False* and assigning a handler for the *OnLogin* event creates a situation where it is impossible to log in to the database because the default dialog does not appear and the *OnLogin* event handler never executes.
  - For *TADOConnection*, the event is an *OnWillConnect* event. The event handler has five parameters, the connection component and four parameters that return values to influence the connection (including two for user name and password). This event always occurs, regardless of the value of *LoginPrompt*.

Write an event handler for the event in which you set the login parameters. Here is an example where the values for the USER NAME and PASSWORD parameters are provided from a global variable (*UserName*) and a method that returns a password given a user name (*PasswordSearch*)

```
procedure TForm1.Database1Login(Database: TDatabase; LoginParams: TStrings);
begin
  LoginParams.Values['USER NAME'] := UserName;
  LoginParams.Values['PASSWORD'] := PasswordSearch(UserName);
end;
```

As with the other methods of providing login parameters, when writing an *OnLogin* or *OnWillConnect* event handler, avoid hard coding the password in your application code. It should appear only as an encrypted value, an entry in a secure database your application uses to look up the value, or be dynamically obtained from the user.

## Managing transactions

A *transaction* is a group of actions that must all be carried out successfully on one or more tables in a database before they are *committed* (made permanent). If one of the actions in the group fails, then all actions are *rolled back* (undone). By using transactions, you ensure that the database is not left in an inconsistent state when a problem occurs completing one of the actions that make up the transaction.

For example, in a banking application, transferring funds from one account to another is an operation you would want to protect with a transaction. If, after decrementing the balance in one account, an error occurred incrementing the balance in the other, you want to roll back the transaction so that the database still reflects the correct total balance.

It is always possible to manage transactions by sending SQL commands directly to the database. Most databases provide their own transaction management model, although some have no transaction support at all. For servers that support it, you may want to code your own transaction management directly, taking advantage of advanced transaction management capabilities on a particular database server, such as schema caching.

If you do not need to use any advanced transaction management capabilities, connection components provide a set of methods and properties you can use to manage transactions without explicitly sending any SQL commands. Using these properties and methods has the advantage that you do not need to customize your application for each type of database server you use, as long as the server supports transactions. (The BDE also provides limited transaction support for local tables with no server transaction support. When not using the BDE, trying to start transactions on a database that does not support them causes connection components to raise an exception.)

**Warning** When a dataset provider component applies updates, it implicitly generates transactions for any updates. Be careful that any transactions you explicitly start do not conflict with those generated by the provider.

## Starting a transaction

When you start a transaction, all subsequent statements that read from or write to the database occur in the context of that transaction, until the transaction is explicitly terminated or (in the case of overlapping transactions) until another transaction is started. Each statement is considered part of a group. Changes must be successfully committed to the database, or every change made in the group must be undone.

While the transaction is in process, your view of the data in database tables is determined by your transaction isolation level. For information about transaction isolation levels, see "Specifying the transaction isolation level" on page 23-9.

For *TADOConnection*, start a transaction by calling the *BeginTrans* method:

```
Level := ADOConnection1.BeginTrans;
```

*BeginTrans* returns the level of nesting for the transaction that started. A nested transaction is one that is nested within another, parent, transaction. After the server starts the transaction, the ADO connection receives an *OnBeginTransComplete* event.

For *TDatabase,* use the *StartTransaction* method instead. *TDataBase* does not support nested or overlapped transactions: If you call a *TDatabase* component's *StartTransaction* method while another transaction is underway, it raises an exception. To avoid calling *StartTransaction*, you can check the *InTransaction* property:

```
if not Database1.InTransaction then
  Database1.StartTransaction;
```

*TSQLConnection* also uses the *StartTransaction* method, but it uses a version that gives you a lot more control. Specifically, *StartTransaction* takes a transaction descriptor, which lets you manage multiple simultaneous transactions and specify the transaction isolation level on a per-transaction basis. (For more information on transaction levels, see "Specifying the transaction isolation level" on page 23-9.) In order to manage multiple simultaneous transactions, set the *TransactionID* field of the transaction descriptor to a unique value. *TransactionID* can be any value you choose, as long as it is unique (does not conflict with any other transaction currently underway). Depending on the server, transactions started by *TSQLConnection* can be nested (as they can be when using ADO) or they can be overlapped.

```
var
  TD: TTransactionDesc;
begin
  TD.TransactionID := 1;
  TD.IsolationLevel := xilREADCOMMITTED;
  SQLConnection1.StartTransaction(TD);
```

By default, with overlapped transactions, the first transaction becomes inactive when the second transaction starts, although you can postpone committing or rolling back the first transaction until later. If you are using *TSQLConnection* with an InterBase database, you can identify each dataset in your application with a particular active transaction, by setting its *TransactionLevel* property. That is, after starting a second transaction, you can continue to work with both transactions simultaneously, simply by associating a dataset with the transaction you want.

**Note** Unlike *TADOConnection*, *TSQLConnection* and *TDatabase* do not receive any events when the transactions starts.

InterBase express offers you even more control than *TSQLConnection* by using a separate transaction component rather than starting transactions using the connection component. You can, however, use *TIBDatabase* to start a default transaction:

```
if not IBDatabase1.DefaultTransaction.InTransaction then
  IBDatabase1.DefaultTransaction.StartTransaction;
```

You can have overlapped transactions by using two separate transaction components. Each transaction component has a set of parameters that let you configure the transaction. These let you specify the transaction isolation level, as well as other properties of the transaction.

## Ending a transaction

Ideally, a transaction should only last as long as necessary. The longer a transaction is active, the more simultaneous users that access the database, and the more concurrent, simultaneous transactions that start and end during the lifetime of your transaction, the greater the likelihood that your transaction will conflict with another when you attempt to commit any changes.

### Ending a successful transaction

When the actions that make up the transaction have all succeeded, you can make the database changes permanent by committing the transaction. For *TDatabase*, you commit a transaction using the *Commit* method:

```
MyOracleConnection.Commit;
```

For *TSQLConnection*, you also use the *Commit* method, but you must specify which transaction you are committing by supplying the transaction descriptor you gave to the *StartTransaction* method:

```
MyOracleConnection.Commit(TD);
```

For *TIBDatabase*, you commit a transaction object using its *Commit* method:

```
IBDatabase1.DefaultTransaction.Commit;
```

For *TADOConnection*, you commit a transaction using the *CommitTrans* method:

```
ADOConnection1.CommitTrans;
```

**Note** It is possible for a nested transaction to be committed, only to have the changes rolled back later if the parent transaction is rolled back.

After the transaction is successfully committed, an ADO connection component receives an *OnCommitTransComplete* event. Other connection components do not receive any similar events.

A call to commit the current transaction is usually attempted in a **try...except** statement. That way, if the transaction cannot commit successfully, you can use the **except** block to handle the error and retry the operation or to roll back the transaction.

### Ending an unsuccessful transaction

If an error occurs when making the changes that are part of the transaction or when trying to commit the transaction, you will want to discard all changes that make up the transaction. Discarding these changes is called rolling back the transaction.

For *TDatabase*, you roll back a transaction by calling the *Rollback* method:

```
MyOracleConnection.Rollback;
```

For *TSQLConnection*, you also use the *Rollback* method, but you must specify which transaction you are rolling back by supplying the transaction descriptor you gave to the *StartTransaction* method:

```
MyOracleConnection.Rollback(TD);
```

For *TIBDatabase*, you roll back a transaction object by calling its *Rollback* method:

```
IBDatabase1.DefaultTransaction.Rollback;
```

For *TADOConnection*, you roll back a transaction by calling the *RollbackTrans* method:

```
ADOConnection1.RollbackTrans;
```

After the transaction is successfully rolled back, an ADO connection component receives an *OnRollbackTransComplete* event. Other connection components do not receive any similar events.

A call to roll back the current transaction usually occurs in

- Exception handling code when you can’t recover from a database error.
- Button or menu event code, such as when a user clicks a Cancel button.

## Specifying the transaction isolation level

Transaction isolation level determines how a transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction “sees” of other transactions’ changes to a table.

Each server type supports a different set of possible transaction isolation levels. There are three possible transaction isolation levels:

- *DirtyRead*: When the isolation level is *DirtyRead*, your transaction sees all changes made by other transactions, even if they have not been committed. Uncommitted changes are not permanent, and might be rolled back at any time. This value provides the least isolation, and is not available for many database servers (such as Oracle, Sybase, MS-SQL, and InterBase).

- *ReadCommitted*: When the isolation level is *ReadCommitted*, only committed changes made by other transactions are visible. Although this setting protects your transaction from seeing uncommitted changes that may be rolled back, you may still receive an inconsistent view of the database state if another transaction is committed while you are in the process of reading. This level is available for all transactions except local transactions managed by the BDE.
- *RepeatableRead*: When the isolation level is *RepeatableRead*, your transaction is guaranteed to see a consistent state of the database data. Your transaction sees a single snapshot of the data. It cannot see any subsequent changes to data by other simultaneous transactions, even if they are committed. This isolation level guarantees that once your transaction reads a record, its view of that record will not change. At this level your transaction is most isolated from changes made by other transactions. This level is not available on some servers, such as Sybase and MS-SQL and is unavailable on local transactions managed by the BDE.

In addition, *TSQLConnection* lets you specify database-specific custom isolation levels. Custom isolation levels are defined by the *dbExpress* driver. See your driver documentation for details.

**Note** For a detailed description of how each isolation level is implemented, see your server documentation.

*TDatabase* and *TADOConnection* let you specify the transaction isolation level by setting the *TransIsolation* property. When you set *TransIsolation* to a value that is not supported by the database server, you get the next highest level of isolation (if available). If there is no higher level available, the connection component raises an exception when you try to start a transaction.

When using *TSQLConnection*, transaction isolation level is controlled by the *IsolationLevel* field of the transaction descriptor.

When using InterBase express, transaction isolation level is controlled by a transaction parameter.

## Sending commands to the server

All database connection components except *TIBDatabase* let you execute SQL statements on the associated server by calling the *Execute* method. Although *Execute* can return a cursor when the statement is a SELECT statement, this use is not recommended. The preferred method for executing statements that return data is to use a dataset.

The *Execute* method is very convenient for executing simple SQL statements that do not return any records. Such statements include Data Definition Language (DDL) statements, which operate on or create a database's metadata, such as CREATE INDEX, ALTER TABLE, and DROP DOMAIN. Some Data Manipulation Language (DML) SQL statements also do not return a result set. The DML statements that perform an action on data but do not return a result set are: INSERT, DELETE, and UPDATE.

The syntax for the *Execute* method varies with the connection type:

- For *TDatabase*, *Execute* takes four parameters: a string that specifies a single SQL statement that you want to execute, a *TParams* object that supplies any parameter values for that statement, a boolean that indicates whether the statement should be cached because you will call it again, and a pointer to a BDE cursor that can be returned (It is recommended that you pass nil).
- For *TADOConnection*, there are two versions of *Execute.* The first takes a WideString that specifies the SQL statement and a second parameter that specifies a set of options that control whether the statement is executed asynchronously and whether it returns any records. This first syntax returns an interface for the returned records. The second syntax takes a WideString that specifies the SQL statement, a second parameter that returns the number of records affected when the statement executes, and a third that specifies options such as whether the statement executes asynchronously. Note that neither syntax provides for passing parameters.
- For *TSQLConnection*, *Execute* takes three parameters: a string that specifies a single SQL statement that you want to execute, a *TParams* object that supplies any parameter values for that statement, and a pointer that can receive a *TCustomSQLDataSet* that is created to return records.

**Note** *Execute* can only execute one SQL statement at a time. It is not possible to execute multiple SQL statements with a single call to *Execute*, as you can with SQL scripting utilities. To execute more than one statement, call *Execute* repeatedly.

It is relatively easy to execute a statement that does not include any parameters. For example, the following code executes a CREATE TABLE statement (DDL) without any parameters on a *TSQLConnection* component:

```
procedure TForm1.CreateTableButtonClick(Sender: TObject);
var
  SQLstmt: String;
begin
  SQLConnection1.Connected := True;
  SQLstmt := 'CREATE TABLE NewCusts ' +
    '( ' +
    '  CustNo INTEGER, ' +
    '  Company CHAR(40), ' +
    '  State CHAR(2), ' +
    '  PRIMARY KEY (CustNo) ' +
    ')';
  SQLConnection1.Execute(SQLstmt, nil, nil);
end;
```

To use parameters, you must create a *TParams* object. For each parameter value, use the *TParams.CreateParam* method to add a *TParam* object. Then use properties of *TParam* to describe the parameter and set its value.

This process is illustrated in the following example, which uses *TDatabase* to execute an INSERT statement. The INSERT statement has a single parameter named: *StateParam*. A *TParams* object (called *stmtParams*) is created to supply a value of “CA” for that parameter.

```
procedure TForm1.INSERT_WithParamsButtonClick(Sender: TObject);
var
  SQLstmt: String;
  stmtParams: TParams;
begin
  stmtParams := TParams.Create;
  try
    Database1.Connected := True;
    stmtParams.CreateParam(ftString, 'StateParam', ptInput);
    stmtParams.ParamByName('StateParam').AsString := 'CA';
    SQLstmt := 'INSERT INTO "Custom.db" '+
      '(CustNo, Company, State) ' +
      'VALUES (7777, "Robin Dabank Consulting", :StateParam)';
    Database1.Execute(SQLstmt, stmtParams, False, nil);
  finally
    stmtParams.Free;
  end;
end;
```

If the SQL statement includes a parameter but you do not supply a *TParam* object to provide its value, the SQL statement may cause an error when executed (this depends on the particular database back-end used). If a *TParam* object is provided but there is no corresponding parameter in the SQL statement, an exception is raised when the application attempts to use the *TParam*.

## Working with associated datasets

All database connection components maintain a list of all datasets that use them to connect to a database. A connection component uses this list, for example, to close all of the datasets when it closes the database connection.

You can use this list as well, to perform actions on all the datasets that use a specific connection component to connect to a particular database.

### Closing all datasets without disconnecting from the server

The connection component automatically closes all datasets when you close its connection. There may be times, however, when you want to close all datasets without disconnecting from the database server.

To close all open datasets without disconnecting from a server, you can use the *CloseDataSets* method.

For *TADOConnection* and *TIBDatabase*, calling *CloseDataSets* always leaves the connection open. For *TDatabase* and *TSQLConnection*, you must also set the *KeepConnection* property to *True*.

### Iterating through the associated datasets

To perform any actions (other than closing them all) on all the datasets that use a connection component, use the *DataSets* and *DataSetCount* properties. *DataSets* is an indexed array of all datasets that are linked to the connection component. For all connection components except *TADOConnection*, this list includes only the active datasets. *TADOConnection* lists the inactive datasets as well. *DataSetCount* is the number of datasets in this array.

**Note** When you use a specialized client dataset to cache updates (as opposed to the generic client dataset, *TClientDataSet*), the *DataSets* property lists the internal dataset owned by the client dataset, not the client dataset itself.

You can use *DataSets* with *DataSetCount* to cycle through all currently active datasets in code. For example, the following code cycles through all active datasets and disables any controls that use the data they provide:

```
var
  I: Integer;
begin
  with MyDBConnection do
  begin
    for I := 0 to DataSetCount - 1 do
      DataSets[I].DisableControls;
  end;
end;
```

**Note** *TADOConnection* supports command objects as well as datasets. You can iterate through these much like you iterate through the datasets, by using the *Commands* and *CommandCount* properties.

## Obtaining metadata

All database connection components can retrieve lists of metadata on the database server, although they vary in the types of metadata they retrieve. The methods that retrieve metadata fill a string list with the names of various entities available on the server. You can then use this information, for example, to let your users dynamically select a table at runtime.

You can use a *TADOConnection* component to retrieve metadata about the tables and stored procedures available on the ADO data store. You can then use this information, for example, to let your users dynamically select a table or stored procedure at runtime.

## Listing available tables

The *GetTableNames* method copies a list of table names to an already-existing string list object. This can be used, for example, to fill a list box with table names that the user can then use to choose a table to open. The following line fills a listbox with the names of all tables on the database:

```
MyDBConnection.GetTableNames(ListBox1.Items, False);
```

*GetTableNames* has two parameters: the string list to fill with table names, and a boolean that indicates whether the list should include system tables, or ordinary tables. Note that not all servers use system tables to store metadata, so asking for system tables may result in an empty list.

**Note** For most database connection components, *GetTableNames* returns a list of all available non-system tables when the second parameter is *False*. For *TSQLConnection*, however, you have more control over what type is added to the list when you are not fetching only the names of system tables. When using *TSQLConnection*, the types of names added to the list are controlled by the *TableScope* property. *TableScope* indicates whether the list should contain any or all of the following: ordinary tables, system tables, synonyms, and views.

## Listing the fields in a table

The *GetFieldNames* method fills an existing string list with the names of all fields (columns) in a specified table. *GetFieldNames* takes two parameters, the name of the table for which you want to list the fields, and an existing string list to be filled with field names:

```
MyDBConnection.GetFieldNames('Employee', ListBox1.Items);
```

## Listing available stored procedures

To get a listing of all of the stored procedures contained in the database, use the *GetProcedureNames*method. This method takes a single parameter: an already-existing string list to fill:

```
MyDBConnection.GetProcedureNames(ListBox1.Items);
```

**Note** *GetProcedureNames* is only available for *TADOConnection* and *TSQLConnection*.

## Listing available indexes

To get a listing of all indexes defined for a specific table, use the *GetIndexNames* method. This method takes two parameters: the table whose indexes you want, and an already-existing string list to fill:

```
SQLConnection1.GetIndexNames('Employee', ListBox1.Items);
```

**Note** *GetIndexNames* is only available for *TSQLConnection*, although most table-type datasets have an equivalent method.

## Listing stored procedure parameters

To get a list of all parameters defined for a specific stored procedure, use the *GetProcedureParams* method. *GetProcedureParams* fills a *TList* object with pointers to parameter description records, where each record describes a parameter of a specified stored procedure, including its name, index, parameter type, field type, and so on.

*GetProcedureParams* takes two parameters: the name of the stored procedure, and an already-existing *TList* object to fill:

```
SQLConnection1.GetProcedureParams('GetInterestRate', List1);
```

To convert the parameter descriptions that are added to the list into the more familiar *TParams* object, call the global *LoadParamListItems*procedure. Because *GetProcedureParams* dynamically allocates the individual records, your application must free them when it is finished with the information. The global *FreeProcParams* routine can do this for you.

**Note** *GetProcedureParams* is only available for *TSQLConnection*.

Chapter

24

# Understanding datasets

The fundamental unit for accessing data is the dataset family of objects. Your application uses datasets for all database access. A dataset object represents a set of records from a database organized into a logical table. These records may be the records from a single database table, or they may represent the results of executing a query or stored procedure.

All dataset objects that you use in your database applications descend from *TDataSet*, and they inherit data fields, properties, events, and methods from this class. This chapter describes the functionality of *TDataSet* that is inherited by the dataset objects you use in your database applications. You need to understand this shared functionality to use any dataset object.

*TDataSet* is a virtualized dataset, meaning that many of its properties and methods are **virtual** or **abstract**. A *virtual method* is a function or procedure declaration where the implementation of that method can be (and usually is) overridden in descendant objects. An *abstract method* is a function or procedure declaration without an actual implementation. The declaration is a prototype that describes the method (and its parameters and return type, if any) that must be implemented in all descendant dataset objects, but that might be implemented differently by each of them.

Because *TDataSet* contains **abstract** methods, you cannot use it directly in an application without generating a runtime error. Instead, you either create instances of the built-in *TDataSet* descendants and use them in your application, or you derive your own dataset object from *TDataSet* or its descendants and write implementations for all its **abstract** methods.

*TDataSet* defines much that is common to all dataset objects. For example, *TDataSet* defines the basic structure of all datasets: an array of *TField* components that correspond to actual columns in one or more database tables, lookup fields provided by your application, or calculated fields provided by your application. For information about *TField* components, see Chapter 25, "Working with field components."

This chapter describes how to use the common database functionality introduced by *TDataSet*. Bear in mind, however, that although *TDataSet* introduces the methods for this functionality, not all *TDataSet* dependants implement them. In particular, unidirectional datasets implement only a limited subset.

## Using TDataSet descendants

*TDataSet* has several immediate descendants, each of which corresponds to a different data access mechanism. You do not work directly with any of these descendants. Rather, each descendant introduces the properties and methods for using a particular data access mechanism. These properties and methods are then exposed by descendant classes that are adapted to different types of server data. The immediate descendants of *TDataSet* include

- *TBDEDataSet*, which uses the Borland Database Engine (BDE) to communicate with the database server. The *TBDEDataSet* descendants you use are *TTable*, *TQuery*, *TStoredProc*, and *TNestedTable*. The unique features of BDE-enabled datasets are described in Chapter 26, "Using the Borland Database Engine."
- *TCustomADODataSet*, which uses ActiveX Data Objects (ADO) to communicate with an OLEDB data store. The *TCustomADODataSet* descendants you use are *TADODataSet*, *TADOTable*, *TADOQuery*, and *TADOStoredProc*. The unique features of ADO-based datasets are described in Chapter 27, "Working with ADO components."
- *TCustomSQLDataSet*, which uses dbExpress to communicate with a database server. The *TCustomSQLDataSet* descendants you use are *TSQLDataSet*, *TSQLTable*, *TSQLQuery*, and *TSQLStoredProc*. The unique features of dbExpress datasets are described in Chapter 28, "Using unidirectional datasets."
- *TIBCustomDataSet*, which communicates directly with an InterBase database server. The *TIBCustomDataSet* descendants you use are *TIBDataSet*, *TIBTable*, *TIBQuery*, and *TIBStoredProc*.
- *TCustomClientDataSet*, which represents the data from another dataset component or the data from a dedicated file on disk. The *TCustomClientDataSet* descendants you use are *TClientDataSet*, which can connect to an external (source) dataset, and the client datasets that are specialized to a particular data access mechanism (*TBDEClientDataSet, TSimpleDataSet*, and *TIBClientDataSet*), which use an internal source dataset. The unique features of client datasets are described in Chapter 29, "Using client datasets."

Some pros and cons of the various data access mechanisms employed by these *TDataSet* descendants are described in "Using databases" on page 19-1.

In addition to the built-in datasets, you can create your own custom *TDataSet* descendants — for example to supply data from a process other than a database server, such as a spreadsheet. Writing custom datasets allows you the flexibility of managing the data using any method you choose, while still letting you use the VCL data controls to build your user interface. For more information about creating custom components, see the *Component Writer's Guide*, Chapter 1, "Overview of component creation."

Although each *TDataSet* descendant has its own unique properties and methods, some of the properties and methods introduced by descendant classes are the same as those introduced by other descendant classes that use another data access mechanism. For example, there are similarities between the "table" components (*TTable*, *TADOTable*, *TSQLTable*, and *TIBTable*). For information about the commonalities introduced by *TDataSet* descendants, see "Types of datasets" on page 24-24.

## Determining dataset states

The *state*—or *mode*—of a dataset determines what can be done to its data. For example, when a dataset is closed, its state is *dsInactive*, meaning that nothing can be done to its data. At runtime, you can examine a dataset's read-only *State* property to determine its current state. The following table summarizes possible values for the *State* property and what they mean:

**Table 24.1** Values for the dataset State property

| Value | State | Meaning |
|---|---|---|
| *dsInactive* | Inactive | DataSet closed. Its data is unavailable. |
| *dsBrowse* | Browse | DataSet open. Its data can be viewed, but not changed. This is the default state of an open dataset. |
| *dsEdit* | Edit | DataSet open. The current row can be modified. (not supported on unidirectional datasets) |
| *dsInsert* | Insert | DataSet open. A new row is inserted or appended. (not supported on unidirectional datasets) |
| *dsSetKey* | SetKey | DataSet open. Enables setting of ranges and key values used for ranges and *GotoKey* operations. (not supported by all datasets) |
| *dsCalcFields* | CalcFields | DataSet open. Indicates that an *OnCalcFields* event is under way. Prevents changes to fields that are not calculated. |
| *dsCurValue* | CurValue | DataSet open. Indicates that the CurValue property of fields is being fetched for an event handler that responds to errors in applying cached updates. |
| *dsNewValue* | NewValue | DataSet open. Indicates that the NewValue property of fields is being fetched for an event handler that responds to errors in applying cached updates. |
| *dsOldValue* | OldValue | DataSet open. Indicates that the OldValue property of fields is being fetched for an event handler that responds to errors in applying cached updates. |

**Table 24.1** Values for the dataset State property (continued)

| Value | State | Meaning |
|---|---|---|
| *dsFilter* | Filter | DataSet open. Indicates that a filter operation is under way. A restricted set of data can be viewed, and no data can be changed. (not supported on unidirectional datasets) |
| dsBlockRead | Block Read | DataSet open. Data-aware controls are not updated and events are not triggered when the current record changes. |
| dsInternalCalc | Internal Calc | DataSet open. An *OnCalcFields* event is underway for calculated values that are stored with the record. (client datasets only) |
| dsOpening | Opening | DataSet is in the process of opening but has not finished. This state occurs when the dataset is opened for asynchronous fetching. |

Typically, an application checks the dataset state to determine when to perform certain tasks. For example, you might check for the *dsEdit* or *dsInsert* state to ascertain whether you need to post updates.

**Note** Whenever a dataset's state changes, the *OnStateChange* event is called for any data source components associated with the dataset. For more information about data source components and *OnStateChange*, see "Responding to changes mediated by the data source" on page 20-4.

## Opening and closing datasets

To read or write data in a dataset, an application must first open it. You can open a dataset in two ways,

- Set the *Active* property of the dataset to *True*, either at design time in the Object Inspector, or in code at runtime:

```
CustTable.Active := True;
```

- Call the *Open* method for the dataset at runtime,

```
CustQuery.Open;c
```

When you open the dataset, the dataset first receives a *BeforeOpen* event, then it opens a cursor, populating itself with data, and finally, it receives an *AfterOpen* event.

The newly-opened dataset is in browse mode, which means your application can read the data and navigate through it.

You can close a dataset in two ways,

- Set the *Active* property of the dataset to *False*, either at design time in the Object Inspector, or in code at runtime,

```
CustQuery.Active := False;
```

- Call the *Close* method for the dataset at runtime,

```
CustTable.Close;
```

Just as the dataset receives *BeforeOpen* and *AfterOpen* events when you open it, it receives a *BeforeClose* and *AfterClose* event when you close it. handlers that respond to the *Close* method for a dataset. You can use these events, for example, to prompt the user to post pending changes or cancel them before closing the dataset. The following code illustrates such a handler:

```
procedure TForm1.CustTableVerifyBeforeClose(DataSet: TDataSet);
begin
  if (CustTable.State in [dsEdit, dsInsert]) then begin
    case MessageDlg('Post changes before closing?', mtConfirmation, mbYesNoCancel, 0) of
      mrYes:    CustTable.Post;   { save the changes }
      mrNo:     CustTable.Cancel; { abandon the changes}
      mrCancel: Abort;            { abort closing the dataset }
    end;
  end;
end;
```

**Note** You may need to close a dataset when you want to change certain of its properties, such as *TableName* on a *TTable* component. When you reopen the dataset, the new property value takes effect.

## Navigating datasets

Each active dataset has a *cursor*, or pointer, to the current row in the dataset. The *current row* in a dataset is the one whose field values currently show in single-field, data-aware controls on a form, such as *TDBEdit*, *TDBLabel*, and *TDBMemo*. If the dataset supports editing, the current record contains the values that can be manipulated by edit, insert, and delete methods.

You can change the current row by moving the cursor to point at a different row. The following table lists methods you can use in application code to move to different records:

**Table 24.2** Navigational methods of datasets

| Method | Moves the cursor to |
|---|---|
| *First* | The first row in a dataset. |
| *Last* | The last row in a dataset. (not available for unidirectional datasets) |
| *Next* | The next row in a dataset. |
| *Prior* | The previous row in a dataset. (not available for unidirectional datasets) |
| *MoveBy* | A specified number of rows forward or back in a dataset. |

The data-aware, visual component *TDBNavigator* encapsulates these methods as buttons that users can click to move among records at runtime. For information about the navigator component, see "Navigating and manipulating records" on page 20-29.

Whenever you change the current record using one of these methods (or by other methods that navigate based on a search criterion), the dataset receives two events: *BeforeScroll* (before leaving the current record) and *AfterScroll* (after arriving at the new record). You can use these events to update your user interface (for example, to update a status bar that indicates information about the current record).

*TDataSet* also defines two boolean properties that provide useful information when iterating through the records in a dataset.

**Table 24.3** Navigational properties of datasets

| Property | Description |
|---|---|
| *Bof* (Beginning-of-file) | *True*: the cursor is at the first row in the dataset. |
| | *False*: the cursor is not known to be at the first row in the dataset |
| *Eof* (End-of-file) | *True*: the cursor is at the last row in the dataset. |
| | *False*: the cursor is not known to be at the first row in the dataset |

## Using the First and Last methods

The *First* method moves the cursor to the first row in a dataset and sets the *Bof* property to *True*. If the cursor is already at the first row in the dataset, *First* does nothing.

For example, the following code moves to the first record in *CustTable*:

```
CustTable.First;
```

The *Last* method moves the cursor to the last row in a dataset and sets the *Eof* property to *True*. If the cursor is already at the last row in the dataset, *Last* does nothing.

The following code moves to the last record in *CustTable*:

```
CustTable.Last;
```

**Note** The *Last* method raises an exception in unidirectional datasets.

**Tip** While there may be programmatic reasons to move to the first or last rows in a dataset without user intervention, you can also enable your users to navigate from record to record using the *TDBNavigator* component. The navigator component contains buttons that, when active and visible, enable a user to move to the first and last rows of an active dataset. The *OnClick* events for these buttons call the *First* and *Last* methods of the dataset. For more information about making effective use of the navigator component, see "Navigating and manipulating records" on page 20-29.

## Using the Next and Prior methods

The *Next* method moves the cursor forward one row in the dataset and sets the *Bof* property to *False* if the dataset is not empty. If the cursor is already at the last row in the dataset when you call *Next*, nothing happens.

For example, the following code moves to the next record in *CustTable*:

```
CustTable.Next;
```

The *Prior* method moves the cursor back one row in the dataset, and sets *Eof* to *False* if the dataset is not empty. If the cursor is already at the first row in the dataset when you call *Prior*, *Prior* does nothing.

For example, the following code moves to the previous record in *CustTable*:

```
CustTable.Prior;
```

**Note** The *Prior* method raises an exception in unidirectional datasets.

## Using the MoveBy method

*MoveBy* lets you specify how many rows forward or back to move the cursor in a dataset. Movement is relative to the current record at the time that *MoveBy* is called. *MoveBy* also sets the *Bof* and *Eof* properties for the dataset as appropriate.

This function takes an integer parameter, the number of records to move. Positive integers indicate a forward move and negative integers indicate a backward move.

**Note** *MoveBy* raises an exception in unidirectional datasets if you use a negative argument.

*MoveBy* returns the number of rows it moves. If you attempt to move past the beginning or end of the dataset, the number of rows returned by *MoveBy* differs from the number of rows you requested to move. This is because *MoveBy* stops when it reaches the first or last record in the dataset.

The following code moves two records backward in *CustTable*:

```
CustTable.MoveBy(-2);
```

**Note** If your application uses *MoveBy* in a multi-user database environment, keep in mind that datasets are fluid. A record that was five records back a moment ago may now be four, six, or even an unknown number of records back if several users are simultaneously accessing the database and changing its data.

## Using the Eof and Bof properties

Two read-only, runtime properties, *Eof* (End-of-file) and *Bof* (Beginning-of-file), are useful when you want to iterate through all records in a dataset.

### Eof

When *Eof* is *True*, it indicates that the cursor is unequivocally at the last row in a dataset. *Eof* is set to *True* when an application

- Opens an empty dataset.
- Calls a dataset's *Last* method.
- Calls a dataset's *Next* method, and the method fails (because the cursor is currently at the last row in the dataset.
- Calls *SetRange* on an empty range or dataset.

*Eof* is set to *False* in all other cases; you should assume *Eof* is *False* unless one of the conditions above is met *and* you test the property directly.

*Eof* is commonly tested in a loop condition to control iterative processing of all records in a dataset. If you open a dataset containing records (or you call *First*) *Eof* is *False*. To iterate through the dataset a record at a time, create a loop that steps through each record by calling *Next*, and terminates when *Eof* is *True*. *Eof* remains *False* until you call *Next* when the cursor is already on the last record.

The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets Eof False }
  while not CustTable.Eof do { Cycle until Eof is True }
  begin
    { Process each record here }
    f
    CustTable.Next; { Eof False on success; Eof True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

**Tip** This example also shows how to disable and enable data-aware visual controls tied to a dataset. If you disable visual controls during dataset iteration, it speeds processing because your application does not need to update the contents of the controls as the current record changes. After iteration is complete, controls should be enabled again to update them with values for the new current row. Note that enabling of the visual controls takes place in the **finally** clause of a **try...finally** statement. This guarantees that even if an exception terminates loop processing prematurely, controls are not left disabled.

### Bof

When *Bof* is *True*, it indicates that the cursor is unequivocally at the first row in a dataset. *Bof* is set to *True* when an application

- Opens a dataset.
- Calls a dataset's *First* method.
- Calls a dataset's *Prior* method, and the method fails (because the cursor is currently at the first row in the dataset.
- Calls *SetRange* on an empty range or dataset.

*Bof* is set to *False* in all other cases; you should assume *Bof* is *False* unless one of the conditions above is met *and* you test the property directly.

Like *Eof*, *Bof* can be in a loop condition to control iterative processing of records in a dataset. The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

```
CustTable.DisableControls; { Speed up processing; prevent screen flicker }
try
  while not CustTable.Bof do { Cycle until Bof is True }
  begin
    { Process each record here }
    ƒ
    CustTable.Prior; { Bof False on success; Bof True when Prior fails on first record }
  end;
finally
  CustTable.EnableControls; { Display new current row in controls }
end;
```

## Marking and returning to records

In addition to moving from record to record in a dataset (or moving from one record to another by a specific number of records), it is often also useful to mark a particular location in a dataset so that you can return to it quickly when desired. *TDataSet* introduces a bookmarking feature that consists of a *Bookmark* property and five bookmark methods.

*TDataSet* implements **virtual** bookmark methods. While these methods ensure that any dataset object derived from *TDataSet* returns a value if a bookmark method is called, the return values are merely defaults that do not keep track of the current location. *TDataSet* descendants vary in the level of support they provide for bookmarks. None of the dbExpress datasets add any support for bookmarks. ADO datasets can support bookmarks, depending on the underlying database tables. BDE datasets, InterBase express datasets, and client datasets always support bookmarks.

### The Bookmark property

The *Bookmark* property indicates which bookmark among any number of bookmarks in your application is current. *Bookmark* is a string that identifies the current bookmark. Each time you add another bookmark, it becomes the current bookmark.

### The GetBookmark method

To create a bookmark, you must declare a variable of type *TBookmark* in your application, then call *GetBookmark* to allocate storage for the variable and set its value to a particular location in a dataset. The *TBookmark* type is a Pointer.

### The GotoBookmark and BookmarkValid methods

When passed a bookmark, *GotoBookmark* moves the cursor for the dataset to the location specified in the bookmark. Before calling *GotoBookmark*, you can call *BookmarkValid* to determine if the bookmark points to a record. *BookmarkValid* returns *True* if a specified bookmark points to a record.

### The CompareBookmarks method

You can also call *CompareBookmarks* to see if a bookmark you want to move to is different from another (or the current) bookmark. If the two bookmarks refer to the same record (or if both are **nil**), *CompareBookmarks* returns 0.

### The FreeBookmark method

*FreeBookmark* frees the memory allocated for a specified bookmark when you no longer need it. You should also call *FreeBookmark* before reusing an existing bookmark.

### A bookmarking example

The following code illustrates one use of bookmarking:

```
procedure DoSomething (const Tbl: TTable)
var
  Bookmark: TBookmark;
begin
  Bookmark := Tbl.GetBookmark; { Allocate memory and assign a value }
  Tbl.DisableControls; { Turn off display of records in data controls }
  try
    Tbl.First; { Go to first record in table }
    while not Tbl.Eof do {Iterate through each record in table }
    begin
      { Do your processing here }
      …
      Tbl.Next;
    end;
  finally
    Tbl.GotoBookmark(Bookmark);
    Tbl.EnableControls; { Turn on display of records in data controls, if necessary }
    Tbl.FreeBookmark(Bookmark); {Deallocate memory for the bookmark }
  end;
end;
```

Before iterating through records, controls are disabled. Should an error occur during iteration through records, the **finally** clause ensures that controls are always enabled and that the bookmark is always freed even if the loop terminates prematurely.

## Searching datasets

If a dataset is not unidirectional, you can search against it using the *Locate* and *Lookup* methods. These methods enable you to search on any type of columns in any dataset.

**Note** Some *TDataSet* descendants introduce an additional family of methods for searching based on an index. For information about these additional methods, see "Using Indexes to search for records" on page 24-28.

### Using Locate

*Locate* moves the cursor to the first row matching a specified set of search criteria. In its simplest form, you pass *Locate* the name of a column to search, a field value to match, and an options flag specifying whether the search is case-insensitive or if it can use partial-key matching. (Partial-key matching is when the criterion string need only be a prefix of the field value.) For example, the following code moves the cursor to the first row in the *CustTable* where the value in the *Company* column is "Professional Divers, Ltd.":

```
var
  LocateSuccess: Boolean;
  SearchOptions: TLocateOptions;
begin
  SearchOptions := [loPartialKey];
  LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.', SearchOptions);
end;
```

If *Locate* finds a match, the first record containing the match becomes the current record. *Locate* returns *True* if it finds a matching record, *False* if it does not. If a search fails, the current record does not change.

The real power of *Locate* comes into play when you want to search on multiple columns and specify multiple values to search for. Search values are Variants, which means you can specify different data types in your search criteria. To specify multiple columns in a search string, separate individual items in the string with semicolons.

Because search values are Variants, if you pass multiple values, you must either pass a Variant array as an argument (for example, the return values from the *Lookup* method), or you must construct the Variant array in code using the *VarArrayOf* function. The following code illustrates a search on multiple columns using multiple search values and partial-key matching:

```
with CustTable do
  Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver','P']), loPartialKey);
```

*Locate* uses the fastest possible method to locate matching records. If the columns to search are indexed and the index is compatible with the search options you specify, *Locate* uses the index.

## Using Lookup

*Lookup* searches for the first row that matches specified search criteria. If it finds a matching row, it forces the recalculation of any calculated fields and lookup fields associated with the dataset, then returns one or more fields from the matching row. *Lookup* does not move the cursor to the matching row; it only returns values from it.

In its simplest form, you pass *Lookup* the name of field to search, the field value to match, and the field or fields to return. For example, the following code looks for the first record in the *CustTable* where the value of the *Company* field is “Professional Divers, Ltd.”, and returns the company name, a contact person, and a phone number for the company:

```
var
  LookupResults: Variant;
begin
  LookupResults := CustTable.Lookup('Company', 'Professional Divers, Ltd.',
     'Company;Contact; Phone');
end;
```

*Lookup* returns values for the specified fields from the first matching record it finds. Values are returned as Variants. If more than one return value is requested, *Lookup* returns a Variant array. If there are no matching records, *Lookup* returns a Null Variant. For more information about Variant arrays, see the online Help.

The real power of *Lookup* comes into play when you want to search on multiple columns and specify multiple values to search for. To specify strings containing multiple columns or result fields, separate individual fields in the string items with semicolons.

Because search values are Variants, if you pass multiple values, you must either pass a Variant array as an argument (for example, the return values from the *Lookup* method), or you must construct the Variant array in code using the *VarArrayOf* function. The following code illustrates a lookup search on multiple columns:

```
var
  LookupResults: Variant;
begin
with CustTable do
  LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver', 'Christiansted']),
    'Company; Addr1; Addr2; State; Zip');
end;
```

Like *Locate*, *Lookup* uses the fastest possible method to locate matching records. If the columns to search are indexed, *Lookup* uses the index.

## Displaying and editing a subset of data using filters

An application is frequently interested in only a subset of records from a dataset. For example, you may be interested in retrieving or viewing only those records for companies based in California in your customer database, or you may want to find a record that contains a particular set of field values. In each case, you can use filters to restrict an application's access to a subset of all records in the dataset.

With unidirectional datasets, you can only limit the records in the dataset by using a query that restricts the records in the dataset. With other *TDataSet* descendants, however, you can define a subset of the data that has already been fetched. To restrict an application's access to a subset of all records in the dataset, you can use filters.

A filter specifies conditions a record must meet to be displayed. Filter conditions can be stipulated in a dataset's *Filter* property or coded into its *OnFilterRecord* event handler. Filter conditions are based on the values in any specified number of fields in a dataset, regardless of whether those fields are indexed. For example, to view only those records for companies based in California, a simple filter might require that records contain a value in the State field of "CA".

**Note** Filters are applied to every record retrieved in a dataset. When you want to filter large volumes of data, it may be more efficient to use a query to restrict record retrieval, or to set a range on an indexed table rather than using filters.

### Enabling and disabling filtering

Enabling filters on a dataset is a three step process:

**1** Create a filter.

**2** Set filter options for string-based filter tests, if necessary.

**3** Set the *Filtered* property to *True*.

When filtering is enabled, only those records that meet the filter criteria are available to an application. Filtering is always a temporary condition. You can turn off filtering by setting the *Filtered* property to *False*.

### Creating filters

There are two ways to create a filter for a dataset:

- Specify simple filter conditions in the *Filter* property. *Filter* is especially useful for creating and applying filters at runtime.
- Write an *OnFilterRecord* event handler for simple or complex filter conditions. With *OnFilterRecord*, you specify filter conditions at design time. Unlike the *Filter* property, which is restricted to a single string containing filter logic, an *OnFilterRecord* event can take advantage of branching and looping logic to create complex, multi-level filter conditions.

The main advantage to creating filters using the *Filter* property is that your application can create, change, and apply filters dynamically, (for example, in response to user input). Its main disadvantages are that filter conditions must be expressible in a single text string, cannot make use of branching and looping constructs, and cannot test or compare its values against values not already in the dataset.

The strengths of the *OnFilterRecord* event are that a filter can be complex and variable, can be based on multiple lines of code that use branching and looping constructs, and can test dataset values against values outside the dataset, such as the text in an edit box. The main weakness of using *OnFilterRecord* is that you set the filter at design time and it cannot be modified in response to user input. (You can, however, create several filter handlers and switch among them in response to general application conditions.)

The following sections describe how to create filters using the *Filter* property and the *OnFilterRecord* event handler.

## Setting the Filter property

To create a filter using the *Filter* property, set the value of the property to a string that contains the filter's test condition. For example, the following statement creates a filter that tests a dataset's *State* field to see if it contains a value for the state of California:

```
Dataset1.Filter := 'State = ' + QuotedStr('CA');
```

You can also supply a value for *Filter* based on text supplied by the user. For example, the following statement assigns the text in from edit box to *Filter*:

```
Dataset1.Filter := Edit1.Text;
```

You can, of course, create a string based on both hard-coded text and user-supplied data:

```
Dataset1.Filter := 'State = ' + QuotedStr(Edit1.Text);
```

Blank field values do not appear unless they are explicitly included in the filter:

```
Dataset1.Filter := 'State <> ‘’CA’’ or State = BLANK';
```

**Note** After you specify a value for *Filter*, to apply the filter to the dataset, set the *Filtered* property to *True*.

Filters can compare field values to literals and to constants using the following comparison and logical operators:

**Table 24.4** Comparison and logical operators that can appear in a filter

| Operator | Meaning |
|---|---|
| < | Less than |
| > | Greater than |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| = | Equal to |

**Table 24.4** Comparison and logical operators that can appear in a filter (continued)

| Operator | Meaning |
|---|---|
| <> | Not equal to |
| AND | Tests two statements are both *True* |
| NOT | Tests that the following statement is not *True* |
| OR | Tests that at least one of two statements is *True* |
| + | Adds numbers, concatenates strings, adds numbers to date/time values (only available for some drivers) |
| - | Subtracts numbers, subtracts dates, or subtracts a number from a date (only available for some drivers) |
| * | Multiplies two numbers (only available for some drivers) |
| / | Divides two numbers (only available for some drivers) |
| * | wildcard for partial comparisons (*FilterOptions* must include *foPartialCompare*) |

By using combinations of these operators, you can create fairly sophisticated filters. For example, the following statement checks to make sure that two test conditions are met before accepting a record for display:

```
(Custno > 1400) AND (Custno < 1500);
```

**Note** When filtering is on, user edits to a record may mean that the record no longer meets a filter's test conditions. The next time the record is retrieved from the dataset, it may therefore "disappear." If that happens, the next record that passes the filter condition becomes the current record.

## Writing an OnFilterRecord event handler

You can write code to filter records using the *OnFilterRecord* events generated by the dataset for each record it retrieves. This event handler implements a test that determines if a record should be included in those that are visible to the application.

To indicate whether a record passes the filter condition, your *OnFilterRecord* handler sets its *Accept* parameter to *True* to include a record, or *False* to exclude it. For example, the following filter displays only those records with the State field set to "CA":

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
  Accept := DataSet['State'].AsString = 'CA';
end;
```

When filtering is enabled, an *OnFilterRecord* event is generated for each record retrieved. The event handler tests each record, and only those that meet the filter's conditions are displayed. Because the *OnFilterRecord* event is generated for every record in a dataset, you should keep the event handler as tightly coded as possible to avoid adversely affecting the performance.

### Switching filter event handlers at runtime

You can code any number of *OnFilterRecord* event handlers and switch among them at runtime. For example, the following statements switch to an *OnFilterRecord* event handler called *NewYorkFilter*:

```
DataSet1.OnFilterRecord := NewYorkFilter;
Refresh;
```

## Setting filter options

The *FilterOptions* property lets you specify whether a filter that compares string-based fields accepts records based on partial comparisons and whether string comparisons are case-sensitive. *FilterOptions* is a set property that can be an empty set (the default), or that can contain either or both of the following values:

**Table 24.5** FilterOptions values

| Value | Meaning |
|---|---|
| *foCaseInsensitive* | Ignore case when comparing strings. |
| *foNoPartialCompare* | Disable partial string matching; that is, don't match strings that end with an asterisk (*). |

For example, the following statements set up a filter that ignores case when comparing values in the *State* field:

```
FilterOptions := [foCaseInsensitive];
Filter := 'State = ' + QuotedStr('CA');
```

## Navigating records in a filtered dataset

There are four dataset methods that navigate among records in a filtered dataset. The following table lists these methods and describes what they do:

**Table 24.6** Filtered dataset navigational methods

| Method | Purpose |
|---|---|
| *FindFirst* | Move to the first record that matches the current filter criteria. The search for the first matching record always begins at the first record in the unfiltered dataset. |
| *FindLast* | Move to the last record that matches the current filter criteria. |
| *FindNext* | Moves from the current record in the filtered dataset to the next one. |
| *FindPrior* | Move from the current record in the filtered dataset to the previous one. |

For example, the following statement finds the first filtered record in a dataset:

```
DataSet1.FindFirst;
```

Provided that you set the *Filter* property or create an *OnFilterRecord* event handler for your application, these methods position the cursor on the specified record regardless of whether filtering is currently enabled. If you call these methods when filtering is not enabled, then they

- Temporarily enable filtering.
- Position the cursor on a matching record if one is found.
- Disable filtering.

**Note** If filtering is disabled and you do not set the *Filter* property or create an *OnFilterRecord* event handler, these methods do the same thing as *First*, *Last*, *Next*, and *Prior*.

All navigational filter methods position the cursor on a matching record (if one is found), make that record the current one, and return *True*. If a matching record is not found, the cursor position is unchanged, and these methods return *False*. You can check the status of the *Found* property to wrap these calls, and only take action when *Found* is *True*. For example, if the cursor is already on the last matching record in the dataset and you call *FindNext*, the method returns *False*, and the current record is unchanged.

## Modifying data

You can use the following dataset methods to insert, update, and delete data if the read-only *CanModify* property is *True*. *CanModify* is *True* unless the dataset is unidirectional, the database underlying the dataset does not permit read and write privileges, or some other factor intervenes. (Intervening factors include the *ReadOnly* property on some datasets or the *RequestLive* property on *TQuery* components.)

**Table 24.7** Dataset methods for inserting, updating, and deleting data

| Method | Description |
|---|---|
| *Edit* | Puts the dataset into *dsEdit* state if it is not already in *dsEdit* or *dsInsert* states. |
| *Append* | Posts any pending data, moves current record to the end of the dataset, and puts the dataset in *dsInsert* state. |
| *Insert* | Posts any pending data, and puts the dataset in *dsInsert* state. |
| *Post* | Attempts to post the new or altered record to the database. If successful, the dataset is put in *dsBrowse* state; if unsuccessful, the dataset remains in its current state. |
| *Cancel* | Cancels the current operation and puts the dataset in *dsBrowse* state. |
| *Delete* | Deletes the current record and puts the dataset in *dsBrowse* state. |

## Editing records

A dataset must be in *dsEdit* mode before an application can modify records. In your code you can use the *Edit* method to put a dataset into *dsEdit* mode if the read-only *CanModify* property for the dataset is *True*.

When a dataset transitions to *dsEdit* mode, it first receives a *BeforeEdit* event. After the transition to edit mode is successfully completed, the dataset receives an *AfterEdit* event. Typically, these events are used for updating the user interface to indicate the current state of the dataset. If the dataset can't be put into edit mode for some reason, an *OnEditError* event occurs, where you can inform the user of the problem or try to correct the situation that prevented the dataset from entering edit mode.

On forms in your application, some data-aware controls can automatically put a dataset into *dsEdit* state if

- The control's *ReadOnly* property is *False* (the default),
- The *AutoEdit* property of the data source for the control is *True*, and
- *CanModify* is *True* for the dataset.

**Note** Even if a dataset is in *dsEdit* state, editing records may not succeed for SQL-based databases if your application's user does not have proper SQL access privileges.

Once a dataset is in *dsEdit* mode, a user can modify the field values for the current record that appears in any data-aware controls on a form. Data-aware controls for which editing is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

If you have a navigator component on your form, users can cancel edits by clicking the navigator's Cancel button. Canceling edits returns a dataset to *dsBrowse* state.

In code, you must write or cancel edits by calling the appropriate methods. You write changes by calling *Post*. You cancel them by calling *Cancel*. In code, *Edit* and *Post* are often used together. For example,

```
with CustTable do
begin
  Edit;
  FieldValues['CustNo'] := 1234;
  Post;
end;
```

In the previous example, the first line of code places the dataset in *dsEdit* mode. The next line of code assigns the number 1234 to the *CustNo* field of the current record. Finally, the last line writes, or posts, the modified record. If you are not caching updates, posting writes the change back to the database. If you are caching updates, the change is written to a temporary buffer, where it stays until the dataset's *ApplyUpdates* method is called.

## Adding new records

A dataset must be in *dsInsert* mode before an application can add new records. In code, you can use the *Insert* or *Append* methods to put a dataset into *dsInsert* mode if the read-only *CanModify* property for the dataset is *True*.

When a dataset transitions to *dsInsert* mode, it first receives a *BeforeInsert* event. After the transition to insert mode is successfully completed, the dataset receives first an *OnNewRecord* event hand then an *AfterInsert* event. You can use these events, for example, to provide initial values to newly inserted records:

```
procedure TForm1.OrdersTableNewRecord(DataSet: TDataSet);
begin
  DataSet.FieldByName('OrderDate').AsDateTime := Date;
end;
```

On forms in your application, the data-aware grid and navigator controls can put a dataset into *dsInsert* state if

- The control's *ReadOnly* property is *False* (the default), and
- *CanModify* is *True* for the dataset.

**Note** Even if a dataset is in *dsInsert* state, adding records may not succeed for SQL-based databases if your application's user does not have proper SQL access privileges.

Once a dataset is in *dsInsert* mode, a user or application can enter values into the fields associated with the new record. Except for the grid and navigational controls, there is no visible difference to a user between *Insert* and *Append*. On a call to *Insert*, an empty row appears in a grid above what was the current record. On a call to *Append*, the grid is scrolled to the last record in the dataset, an empty row appears at the bottom of the grid, and the *Next* and *Last* buttons are dimmed on any navigator component associated with the dataset.

Data-aware controls for which inserting is enabled automatically call *Post* when a user executes any action that changes which record is current (such as moving to a different record in a grid). Otherwise you must call *Post* in your code.

*Post* writes the new record to the database, or, if you are caching updates, *Post* writes the record to an in-memory cache. To write cached inserts and appends to the database, call the dataset's *ApplyUpdates* method.

### Inserting records

*Insert* opens a new, empty record before the current record, and makes the empty record the current record so that field values for the record can be entered either by a user or by your application code.

When an application calls *Post* (or *ApplyUpdates* when using cached updates), a newly inserted record is written to a database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed Paradox and dBASE tables, the record is inserted into the dataset at its current position.

- For SQL databases, the physical location of the insertion is implementation-specific. If the table is indexed, the index is updated with the new record information.

### Appending records

*Append* opens a new, empty record at the end of the dataset, and makes the empty record the current one so that field values for the record can be entered either by a user or by your application code.

When an application calls *Post* (or *ApplyUpdates* when using cached updates), a newly appended record is written to a database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed Paradox and dBASE tables, the record is added to the end of the dataset.
- For SQL databases, the physical location of the append is implementation-specific. If the table is indexed, the index is updated with the new record information.

## Deleting records

Use the *Delete* method to delete the current record in an active dataset. When the *Delete* method is called,

- The dataset receives a *BeforeDelete* event.
- The dataset attempts to delete the current record.
- The dataset returns to the *dsBrowse* state.
- The dataset receives an *AfterDelete* event.

If want to prevent the deletion in the *BeforeDelete* event handler, you can call the global *Abort* procedure:

```
procedure TForm1.TableBeforeDelete (Dataset: TDataset)
begin
  if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel, 0) <> mrYes then
    Abort;
end;
```

If *Delete* fails, it generates an *OnDeleteError* event. If the *OnDeleteError* event handler can't correct the problem, the dataset remains in *dsEdit* state. If *Delete* succeeds, the dataset reverts to the *dsBrowse* state and the record that followed the deleted record becomes the current record.

If you are caching updates, the deleted record is not removed from the underlying database table until you call *ApplyUpdates*.

If you provide a navigator component on your forms, users can delete the current record by clicking the navigator's Delete button. In code, you must call *Delete* explicitly to remove the current record.

## Posting data

After you finish editing a record, you must call the *Post* method to write out your changes. The *Post* method behaves differently, depending on the dataset's state and on whether you are caching updates.

- If you are not caching updates, and the dataset is in the *dsEdit* or *dsInsert* state, *Post* writes the current record to the database and returns the dataset to the *dsBrowse* state.
- If you are caching updates, and the dataset is in the *dsEdit* or *dsInsert* state, *Post* writes the current record to an internal cache and returns the dataset to the *dsBrowse* state. The edits are net written to the database until you call *ApplyUpdates*.
- If the dataset is in the *dsSetKey* state, *Post* returns the dataset to the *dsBrowse* state.

Regardless of the initial state of the dataset, *Post* generates *BeforePost* and *AfterPost* events, before and after writing the current changes. You can use these events to update the user interface, or prevent the dataset from posting changes by calling the *Abort* procedure. If the call to *Post* fails, the dataset receives an *OnPostError* event, where you can inform the user of the problem or attempt to correct it.

Posting can be done explicitly, or implicitly as part of another procedure. When an application moves off the current record, *Post* is called implicitly. Calls to the *First*, *Next*, *Prior*, and *Last* methods perform a *Post* if the table is in *dsEdit* or *dsInsert* modes. The *Append* and *Insert* methods also implicitly post any pending data.

**Warning** The *Close* method does not call *Post* implicitly. Use the *BeforeClose* event to post any pending edits explicitly.

## Canceling changes

An application can undo changes made to the current record at any time, if it has not yet directly or indirectly called *Post*. For example, if a dataset is in *dsEdit* mode, and a user has changed the data in one or more fields, the application can return the record back to its original values by calling the *Cancel* method for the dataset. A call to *Cancel* always returns a dataset to *dsBrowse* state.

If the dataset was in *dsEdit* or *dsInsert* mode when your application called *Cancel*, it receives *BeforeCancel* and *AfterCancel* events before and after the current record is restored to its original values.

On forms, you can allow users to cancel edit, insert, or append operations by including the Cancel button on a navigator component associated with the dataset, or you can provide code for your own Cancel button on the form.

## Modifying entire records

On forms, all data-aware controls except for grids and the navigator provide access to a single field in a record.

In code, however, you can use the following methods that work with entire record structures provided that the structure of the database tables underlying the dataset is stable and does not change. The following table summarizes the methods available for working with entire records rather than individual fields in those records:

**Table 24.8** Methods that work with entire records

| Method | Description |
|---|---|
| *AppendRecord*([array of values]) | Appends a record with the specified column values at the end of a table; analogous to *Append*. Performs an implicit *Post*. |
| *InsertRecord*([array of values]) | Inserts the specified values as a record before the current cursor position of a table; analogous to *Insert*. Performs an implicit *Post*. |
| *SetFields*([array of values]) | Sets the values of the corresponding fields; analogous to assigning values to *TField*s. The application must perform an explicit *Post*. |

These method take an array of values as an argument, where each value corresponds to a column in the underlying dataset. The values can be literals, variables, or NULL. If the number of values in an argument is less than the number of columns in a dataset, then the remaining values are assumed to be NULL.

For unindexed datasets, *AppendRecord* adds a record to the end of the dataset and *InsertRecord* inserts a record after the current cursor position. For indexed datasets, both methods place the record in the correct position in the table, based on the index. In both cases, the methods move the cursor to the record's position.

*SetFields* assigns the values specified in the array of parameters to fields in the dataset. To use *SetFields*, an application must first call *Edit* to put the dataset in *dsEdit* mode. To apply the changes to the current record, it must perform a *Post*.

If you use *SetFields* to modify some, but not all fields in an existing record, you can pass NULL values for fields you do not want to change. If you do not supply enough values for all fields in a record, SetFields assigns NULL values to them. NULL values overwrite any existing values already in those fields.

For example, suppose a database has a COUNTRY table with columns for Name, Capital, Continent, Area, and Population. If a *TTable* component called *CountryTable* were linked to the COUNTRY table, the following statement would insert a record into the COUNTRY table:

```
CountryTable.InsertRecord(['Japan', 'Tokyo', 'Asia']);
```

This statement does not specify values for Area and Population, so NULL values are inserted for them. The table is indexed on Name, so the statement would insert the record based on the alphabetic collation of "Japan".

To update the record, an application could use the following code:

```
with CountryTable do
begin
  if Locate('Name', 'Japan', loCaseInsensitive) then;
  begin
    Edit;
    SetFields(nil, nil, nil, 344567, 164700000);
    Post;
  end;
end;
```

This code assigns values to the Area and Population fields and then posts them to the database. The three NULL pointers act as place holders for the first three columns to preserve their current contents.

## Calculating fields

Using the Fields editor, you can define calculated fields for your datasets. When a dataset contains calculated fields, you provide the code to calculate those field's values in an *OnCalcFields* event handler. For details on how to define calculated fields using the Fields editor, see "Defining a calculated field" on page 25-7.

The *AutoCalcFields* property determines when *OnCalcFields* is called. If *AutoCalcFields* is *True*, *OnCalcFields* is called when

- A dataset is opened.
- The dataset enters edit mode.
- A record is retrieved from the database.
- Focus moves from one visual component to another, or from one column to another in a data-aware grid control and the current record has been modified.

If *AutoCalcFields* is *False*, then *OnCalcFields* is not called when individual fields within a record are edited (the fourth condition above).

**Caution** *OnCalcFields* is called frequently, so the code you write for it should be kept short. Also, if *AutoCalcFields* is *True*, *OnCalcFields* should not perform any actions that modify the dataset (or a linked dataset if it is part of a master-detail relationship), because this leads to recursion. For example, if *OnCalcFields* performs a *Post*, and *AutoCalcFields* is *True*, then *OnCalcFields* is called again, causing another *Post*, and so on.

When *OnCalcFields* executes, a dataset enters *dsCalcFields* mode. This state prevents modifications or additions to the records except for the calculated fields the handler is designed to modify. The reason for preventing other modifications is because *OnCalcFields* uses the values in other fields to derive calculated field values. Changes to those other fields might otherwise invalidate the values assigned to calculated fields. After *OnCalcFields* is completed, the dataset returns to *dsBrowse* state.

## Types of datasets

"Using TDataSet descendants" on page 24-2 classifies *TDataSet* descendants by the method they use to access their data. Another useful way to classify *TDataSet* descendants is to consider the type of server data they represent. Viewed this way, there are three basic classes of datasets:

- **Table type datasets**: Table type datasets represent a single table from the database server, including all of its rows and columns. Table type datasets include *TTable*, *TADOTable*, *TSQLTable*, and *TIBTable*.

  Table type datasets let you take advantage of indexes defined on the server. Because there is a one-to-one correspondence between database table and dataset, you can use server indexes that are defined for the database table. Indexes allow your application to sort the records in the table, speed searches and lookups, and can form the basis of a master/detail relationship. Some table type datasets also take advantage of the one-to-one relationship between dataset and database table to let you perform table-level operations such as creating and deleting database tables.

- **Query-type datasets**: Query-type datasets represent a single SQL command, or query. Queries can represent the result set from executing a command (typically a SELECT statement), or they can execute a command that does not return any records (for example, an UPDATE statement). Query-type datasets include *TQuery*, *TADOQuery*, *TSQLQuery*, and *TIBQuery*.

  To use a query-type dataset effectively, you must be familiar with SQL and your server's SQL implementation, including limitations and extensions to the SQL-92 standard. If you are new to SQL, you may want to purchase a third party book that covers SQL in-depth. One of the best is *Understanding the New SQL: A Complete Guide*, by Jim Melton and Alan R. Simpson, Morgan Kaufmann Publishers.

- **Stored procedure-type datasets**: Stored procedure-type datasets represent a stored procedure on the database server. Stored procedure-type datasets include *TStoredProc*, *TADOStoredProc*, *TSQLStoredProc*, and *TIBStoredProc*.

  A stored procedure is a self-contained program written in the procedure and trigger language specific to the database system used. They typically handle frequently repeated database-related tasks, and are especially useful for operations that act on large numbers of records or that use aggregate or mathematical functions. Using stored procedures typically improves the performance of a database application by:

  - Taking advantage of the server's usually greater processing power and speed.
  - Reducing network traffic by moving processing to the server.

Stored procedures may or may not return data. Those that return data may return it as a cursor (similar to the results of a SELECT query), as multiple cursors (effectively returning multiple datasets), or they may return data in output parameters. These differences depend in part on the server: Some servers do not allow stored procedures to return data, or only allow output parameters. Some servers do not support stored procedures at all. See your server documentation to determine what is available.

**Note** You can usually use a query-type dataset to execute stored procedures because most servers provide extensions to SQL for working with stored procedures. Each server, however, uses its own syntax for this. If you choose to use a query-type dataset instead of a stored procedure-type dataset, see your server documentation for the necessary syntax.

In addition to the datasets that fall neatly into these three categories, *TDataSet* has some descendants that fit into more than one category:

- *TADODataSet* and *TSQLDataSet* have a *CommandType* property that lets you specify whether they represent a table, query, or stored procedure. Property and method names are most similar to query-type datasets, although *TADODataSet* lets you specify an index like a table type dataset.
- *TClientDataSet* represents the data from another dataset. As such, it can represent a table, query, or stored procedure. *TClientDataSet* behaves most like a table type dataset, because of its index support. However, it also has some of the features of queries and stored procedures: the management of parameters and the ability to execute without retrieving a result set.
- Some other client datasets (like *TBDEClientDataSet*) have a *CommandType* property that lets you specify whether they represent a table, query, or stored procedure. Property and method names are like *TClientDataSet*, including parameter support, indexes, and the ability to execute without retrieving a result set.
- *TIBDataSet* can represent both queries and stored procedures. In fact, it can represent multiple queries and stored procedures simultaneously, with separate properties for each.

## Using table type datasets

To use a table type dataset,

1. Place the appropriate dataset component in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Identify the database server that contains the table you want to use. Each table type dataset does this differently, but typically you specify a database connection component:
   - For *TTable*, specify a *TDatabase* component or a BDE alias using the *DatabaseName* property.
   - For *TADOTable*, specify a *TADOConnection* component using the *Connection* property.

- For *TSQLTable*, specify a *TSQLConnection* component using the *SQLConnection* property.
- For *TIBTable*, specify a *TIBConnection* component using the *Database* property.

For information about using database connection components, see Chapter 23, "Connecting to databases."

3 Set the *TableName* property to the name of the table in the database. You can select tables from a drop-down list if you have already identified a database connection component.

4 Place a data source component in the data module or on the form, and set its *DataSet* property to the name of the dataset. The data source component is used to pass a result set from the dataset to data-aware components for display.

## Advantages of using table type datasets

The main advantage of using table type datasets is the availability of indexes. Indexes enable your application to

- Sort the records in the dataset.
- Locate records quickly.
- Limit the records that are visible.
- Establish master/detail relationships.

In addition, the one-to-one relationship between table type datasets and database tables enables many of them to be used for

- Controlling Read/write access to tables
- Creating and deleting tables
- Emptying tables
- Synchronizing tables

## Sorting records with indexes

An index determines the display order of records in a table. Typically, records appear in ascending order based on a primary, or default, index. This default behavior does not require application intervention. If you want a different sort order, however, you must specify either

- An alternate index.
- A list of columns on which to sort (not available on servers that aren't SQL-based).

Indexes let you present the data from a table in different orders. On SQL-based tables, this sort order is implemented by using the index to generate an ORDER BY clause in a query that fetches the table's records. On other tables (such as Paradox and dBASE tables), the index is used by the data access mechanism to present records in the desired order.

## Obtaining information about indexes

You application can obtain information about server-defined indexes from all table type datasets. To obtain a list of available indexes for the dataset, call the *GetIndexNames* method. *GetIndexNames* fills a string list with valid index names. For example, the following code fills a listbox with the names of all indexes defined for the *CustomersTable* dataset:

```
CustomersTable.GetIndexNames(ListBox1.Items);
```

**Note** For Paradox tables, the primary index is unnamed, and is therefore not returned by *GetIndexNames*. You can still change the index back to a primary index on a Paradox table after using an alternative index, however, by setting the *IndexName* property to a blank string.

To obtain information about the fields of the current index, use the

- *IndexFieldCount* property, to determine the number of columns in the index.
- *IndexFields* property, to examine a list the field components for the columns that comprise the index.

The following code illustrates how you might use *IndexFieldCount* and *IndexFields* to iterate through a list of column names in an application:

```
var
  I: Integer;
  ListOfIndexFields: array[0 to 20] of string;
begin
with CustomersTable do
  begin
  for I := 0 to IndexFieldCount - 1 do
     ListOfIndexFields[I] := IndexFields[I].FieldName;
  end;
end;
```

**Note** *IndexFieldCount* is not valid for a dBASE table opened on an expression index.

## Specifying an index with IndexName

Use the *IndexName* property to cause an index to be active. Once active, an index determines the order of records in the dataset. (It can also be used as the basis for a master-detail link, an index-based search, or index-based filtering.)

To activate an index, set the *IndexName* property to the name of the index. In some database systems, primary indexes do not have names. To activate one of these indexes, set *IndexName* to a blank string.

At design-time, you can select an index from a list of available indexes by clicking the property's ellipsis button in the Object Inspector. At runtime set *IndexName* using a ***String*** literal or variable. You can obtain a list of available indexes by calling the *GetIndexNames* method.

The following code sets the index for *CustomersTable* to *CustDescending*:

```
CustomersTable.IndexName := 'CustDescending';
```

### Creating an index with IndexFieldNames

If there is no defined index that implements the sort order you want, you can create a pseudo-index using the *IndexFieldNames* property.

**Note** *IndexName* and *IndexFieldNames* are mutually exclusive. Setting one property clears values set for the other.

The value of *IndexFieldNames* is a string. To specify a sort order, list each column name to use in the order it should be used, and delimit the names with semicolons. Sorting is by ascending order only. Case-sensitivity of the sort depends on the capabilities of your server. See your server documentation for more information.

The following code sets the sort order for *PhoneTable* based on *LastName*, then *FirstName*:

```
PhoneTable.IndexFieldNames := 'LastName;FirstName';
```

**Note** If you use *IndexFieldNames* on Paradox and dBASE tables, the dataset attempts to find an index that uses the columns you specify. If it cannot find such an index, it raises an exception.

## Using Indexes to search for records

You can search against any dataset using the *Locate* and *Lookup* methods of *TDataSet*. However, by explicitly using indexes, some table type datasets can improve over the searching performance provided by the *Locate* and *Lookup* methods.

ADO datasets all support the *Seek* method, which moves to a record based on a set of field values for fields in the current index. *Seek* lets you specify where to move the cursor relative to the first or last matching record.

*TTable* and all types of client dataset support similar indexed-based searches, but use a combination of related methods. The following table summarizes the six related methods provided by *TTable* and client datasets to support index-based searches:

**Table 24.9** Index-based search methods

| Method | Purpose |
|---|---|
| *EditKey* | Preserves the current contents of the search key buffer and puts the dataset into *dsSetKey* state so your application can modify existing search criteria prior to executing a search. |
| *FindKey* | Combines the *SetKey* and *GotoKey* methods in a single method. |
| *FindNearest* | Combines the *SetKey* and *GotoNearest* methods in a single method. |
| *GotoKey* | Searches for the first record in a dataset that exactly matches the search criteria, and moves the cursor to that record if one is found. |
| *GotoNearest* | Searches on string-based fields for the closest match to a record based on partial key values, and moves the cursor to that record. |
| *SetKey* | Clears the search key buffer and puts the table into *dsSetKey* state so your application can specify new search criteria prior to executing a search. |

*GotoKey* and *FindKey* are boolean functions that, if successful, move the cursor to a matching record and return *True*. If the search is unsuccessful, the cursor is not moved, and these functions return *False*.

*GotoNearest* and *FindNearest* always reposition the cursor either on the first exact match found or, if no match is found, on the first record that is greater than the specified search criteria.

## Executing a search with Goto methods

To execute a search using *Goto* methods, follow these general steps:

1. Specify the index to use for the search. This is the same index that sorts the records in the dataset (see "Sorting records with indexes" on page 24-26). To specify the index, use the *IndexName* or *IndexFieldNames* property.
2. Open the dataset.
3. Put the dataset in *dsSetKey* state by calling the *SetKey* method.
4. Specify the value(s) to search on in the *Fields* property. *Fields* is a *TFields* object, which maintains an indexed list of field components you can access by specifying ordinal numbers corresponding to columns. The first column number in a dataset is 0.
5. Search for and move to the first matching record found with *GotoKey* or *GotoNearest*.

For example, the following code, attached to a button's *OnClick* event, uses the *GotoKey* method to move to the first record where the first field in the index has a value that exactly matches the text in an edit box:

```
procedure TSearchDemo.SearchExactClick(Sender: TObject);
begin
  ClientDataSet1.SetKey;
  ClientDataSet1.Fields[0].AsString := Edit1.Text;
  if not ClientDataSet1.GotoKey then
    ShowMessage('Record not found');
end;
```

*GotoNearest* is similar. It searches for the nearest match to a partial field value. It can be used only for string fields. For example,

```
Table1.SetKey;
Table1.Fields[0].AsString := 'Sm';
Table1.GotoNearest;
```

If a record exists with "Sm" as the first two characters of the first indexed field's value, the cursor is positioned on that record. Otherwise, the position of the cursor does not change and *GotoNearest* returns *False*.

### Executing a search with Find methods

The *Find* methods do the same thing as the *Goto* methods, except that you do not need to explicitly put the dataset in *dsSetKey* state to specify the key field values on which to search. To execute a search using *Find* methods, follow these general steps:

1. Specify the index to use for the search. This is the same index that sorts the records in the dataset (see "Sorting records with indexes" on page 24-26). To specify the index, use the *IndexName* or *IndexFieldNames* property.
2. Open the dataset.
3. Search for and move to the first or nearest record with *FindKey* or *FindNearest*. Both methods take a single parameter, a comma-delimited list of field values, where each value corresponds to an indexed column in the underlying table.

**Note** *FindNearest* can only be used for string fields.

### Specifying the current record after a successful search

By default, a successful search positions the cursor on the first record that matches the search criteria. If you prefer, you can set the *KeyExclusive* property to *True* to position the cursor on the next record after the first matching record.

By default, *KeyExclusive* is *False*, meaning that successful searches position the cursor on the first matching record.

### Searching on partial keys

If the dataset has more than one key column, and you want to search for values in a subset of that key, set *KeyFieldCount* to the number of columns on which you are searching. For example, if the dataset's current index has three columns, and you want to search for values using just the first column, set *KeyFieldCount* to 1.

For table type datasets with multiple-column keys, you can search only for values in contiguous columns, beginning with the first. For example, for a three-column key you can search for values in the first column, the first and second, or the first, second, and third, but not just the first and third.

### Repeating or extending a search

Each time you call *SetKey* or *FindKey*, the method clears any previous values in the *Fields* property. If you want to repeat a search using previously set fields, or you want to add to the fields used in a search, call *EditKey* in place of *SetKey* and *FindKey*.

For example, suppose you have already executed a search of the Employee table based on the City field of the "CityIndex" index. Suppose further that "CityIndex" includes both the *City* and *Company* fields. To find a record with a specified company name in a specified city, use the following code:

```
Employee.KeyFieldCount := 2;
Employee.EditKey;
Employee['Company'] := Edit2.Text;
Employee.GotoNearest;
```

## Limiting records with ranges

You can temporarily view and edit a subset of data for any dataset by using filters (see "Displaying and editing a subset of data using filters" on page 24-13). Some table type datasets support an additional way to access a subset of available records, called ranges.

Ranges only apply to *TTable* and to client datasets. Despite their similarities, ranges and filters have different uses. The following topics discuss the differences between ranges and filters and how to use ranges.

### Understanding the differences between ranges and filters

Both ranges and filters restrict visible records to a subset of all available records, but the way they do so differs. A range is a set of contiguously indexed records that fall between specified boundary values. For example, in an employee database indexed on last name, you might apply a range to display all employees whose last names are greater than "Jones" and less than "Smith". Because ranges depend on indexes, you must set the current index to one that can be used to define the range. As with specifying an index to sort records, you can assign the index on which to define a range using either the *IndexName* or the *IndexFieldNames* property.

A filter, on the other hand, is any set of records that share specified data points, regardless of indexing. For example, you might filter an employee database to display all employees who live in California and who have worked for the company for five or more years. While filters can make use of indexes if they apply, filters are not dependent on them. Filters are applied record-by-record as an application scrolls through a dataset.

In general, filters are more flexible than ranges. Ranges, however, can be more efficient when datasets are large and the records of interest to an application are already blocked in contiguously indexed groups. For very large datasets, it may be still more efficient to use the WHERE clause of a query-type dataset to select data. For details on specifying a query, see "Using query-type datasets" on page 24-42.

### Specifying ranges

There are two mutually exclusive ways to specify a range:

- Specify the beginning and ending separately using *SetRangeStart* and *SetRangeEnd*.
- Specify both endpoints at once using *SetRange*.

#### Setting the beginning of a range

Call the *SetRangeStart* procedure to put the dataset into *dsSetKey* state and begin creating a list of starting values for the range. Once you call *SetRangeStart*, subsequent assignments to the *Fields* property are treated as starting index values to use when applying the range. Fields specified must apply to the current index.

For example, suppose your application uses a *TSimpleDataSet* component named *Customers*, linked to the CUSTOMER table, and that you have created persistent field components for each field in the *Customers* dataset. CUSTOMER is indexed on its first column (*CustNo*). A form in the application has two edit components named *StartVal* and *EndVal*, used to specify start and ending values for a range. The following code can be used to create and apply a range:

```
with Customers do
begin
  SetRangeStart;
  FieldByName('CustNo').AsString := StartVal.Text;
  SetRangeEnd;
  if (Length(EndVal.Text) > 0) then
    FieldByName('CustNo').AsString := EndVal.Text;
  ApplyRange;
end;
```

This code checks that the text entered in *EndVal* is not null before assigning any values to *Fields*. If the text entered for *StartVal* is null, then all records from the beginning of the dataset are included, since all values are greater than null. However, if the text entered for *EndVal* is null, then no records are included, since none are less than null.

For a multi-column index, you can specify a starting value for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. If you try to set a value for a field that is not in the index, the dataset raises an exception.

**Tip** To start at the beginning of the dataset, omit the call to *SetRangeStart*.

To finish specifying the start of a range, call *SetRangeEnd* or apply or cancel the range. For information about applying and canceling ranges, see "Applying or canceling a range" on page 24-34.

### Setting the end of a range

Call the *SetRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *SetRangeEnd*, subsequent assignments to the *Fields* property are treated as ending index values to use when applying the range. Fields specified must apply to the current index.

**Warning** Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, Delphi assumes the ending value of the range is a null value. A range with null ending values is always empty.

The easiest way to assign ending values is to call the *FieldByName* method. For example,

```
with Contacts do
begin
  SetRangeStart;
  FieldByName('LastName').AsString := Edit1.Text;
  SetRangeEnd;
  FieldByName('LastName').AsString := Edit2.Text;
  ApplyRange;
end;
```

As with specifying start of range values, if you try to set a value for a field that is not in the index, the dataset raises an exception.

To finish specifying the end of a range, apply or cancel the range. For information about applying and canceling ranges, see "Applying or canceling a range" on page 24-34.

### Setting start- and end-range values

Instead of using separate calls to *SetRangeStart* and *SetRangeEnd* to specify range boundaries, you can call the *SetRange* procedure to put the dataset into *dsSetKey* state and set the starting and ending values for a range with a single call.

*SetRange* takes two constant array parameters: a set of starting values, and a set of ending values. For example, the following statement establishes a range based on a two-column index:

```
SetRange([Edit1.Text, Edit2.Text], [Edit3.Text, Edit4.Text]);
```

For a multi-column index, you can specify starting and ending values for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. To omit a value for the first field in an index, and specify values for successive fields, pass a null value for the omitted field.

Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, the dataset assumes the ending value of the range is a null value. A range with null ending values is always empty because the starting range is greater than or equal to the ending range.

### Specifying a range based on partial keys

If a key is composed of one or more string fields, the *SetRange* methods support partial keys. For example, if an index is based on the *LastName* and *FirstName* columns, the following range specifications are valid:

```
Contacts.SetRangeStart;
Contacts['LastName'] := 'Smith';
Contacts.SetRangeEnd;
Contacts['LastName'] := 'Zzzzzz';
Contacts.ApplyRange;
```

This code includes all records in a range where *LastName* is greater than or equal to "Smith." The value specification could also be:

```
Contacts['LastName'] := 'Sm';
```

This statement includes records that have *LastName* greater than or equal to "Sm."

### Including or excluding records that match boundary values

By default, a range includes all records that are greater than or equal to the specified starting range, and less than or equal to the specified ending range. This behavior is controlled by the *KeyExclusive* property. *KeyExclusive* is *False* by default.

If you prefer, you can set the *KeyExclusive* property for a dataset to *True* to exclude records equal to ending range. For example,

```
Contacts.KeyExclusive := True;
Contacts.SetRangeStart;
Contacts['LastName'] := 'Smith';
Contacts.SetRangeEnd;
Contacts['LastName'] := 'Tyler';
Contacts.ApplyRange;
```

This code includes all records in a range where *LastName* is greater than or equal to “Smith” and less than “Tyler”.

## Modifying a range

Two functions enable you to modify the existing boundary conditions for a range: *EditRangeStart*, for changing the starting values for a range; and *EditRangeEnd*, for changing the ending values for the range.

The process for editing and applying a range involves these general steps:

1. Putting the dataset into *dsSetKey* state and modifying the starting index value for the range.
2. Modifying the ending index value for the range.
3. Applying the range to the dataset.

You can modify either the starting or ending values of the range, or you can modify both boundary conditions. If you modify the boundary conditions for a range that is currently applied to the dataset, the changes you make are not applied until you call *ApplyRange* again.

### Editing the start of a range

Call the *EditRangeStart* procedure to put the dataset into *dsSetKey* state and begin modifying the current list of starting values for the range. Once you call *EditRangeStart*, subsequent assignments to the *Fields* property overwrite the current index values to use when applying the range.

**Tip** If you initially created a start range based on a partial key, you can use *EditRangeStart* to extend the starting value for a range. For more information about ranges based on partial keys, see “Specifying a range based on partial keys” on page 24-33.

### Editing the end of a range

Call the *EditRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *EditRangeEnd*, subsequent assignments to the *Fields* property are treated as ending index values to use when applying the range.

## Applying or canceling a range

When you call *SetRangeStart* or *EditRangeStart* to specify the start of a range, or *SetRangeEnd* or *EditRangeEnd* to specify the end of a range, the dataset enters the *dsSetKey* state. It stays in that state until you apply or cancel the range.

### Applying a range

When you specify a range, the boundary conditions you define are not put into effect until you apply the range. To make a range take effect, call the *ApplyRange* method. *ApplyRange* immediately restricts a user's view of and access to data in the specified subset of the dataset.

### Canceling a range

The *CancelRange* method ends application of a range and restores access to the full dataset. Even though canceling a range restores access to all records in the dataset, the boundary conditions for that range are still available so that you can reapply the range at a later time. Range boundaries are preserved until you provide new range boundaries or modify the existing boundaries. For example, the following code is valid:

```
ƒ
MyTable.CancelRange;
ƒ
{later on, use the same range again. No need to call SetRangeStart, etc.}
MyTable.ApplyRange;
ƒ
```

## Creating master/detail relationships

Table type datasets can be linked into master/detail relationships. When you set up a master/detail relationship, you link two datasets so that all the records of one (the detail) always correspond to the single current record in the other (the master).

Table type datasets support master/detail relationships in two very distinct ways:

- All table type datasets can act as the detail of another dataset by linking cursors. This process is described in "Making the table a detail of another dataset" below.
- *TTable*, *TSQLTable*, and all client datasets can act as the master in a master/detail relationship that uses nested detail tables. This process is described in "Using nested detail tables" on page 24-37.

Each of these approaches has its unique advantages. Linking cursors lets you create master/detail relationships where the master table is any type of dataset. With nested details, the type of dataset that can act as the detail table is limited, but they provide for more options in how to display the data. If the master is a client dataset, nested details provide a more robust mechanism for applying cached updates.

### Making the table a detail of another dataset

A table type dataset's *MasterSource* and *MasterFields* properties can be used to establish one-to-many relationships between two datasets.

The *MasterSource* property is used to specify a data source from which the table gets data from the master table. This data source can be linked to any type of dataset. For instance, by specifying a query's data source in this property, you can link a client dataset as the detail of the query, so that the client dataset tracks events occurring in the query.

The dataset is linked to the master table based on its current index. Before you specify the fields in the master dataset that are tracked by the detail dataset, first specify the index in the detail dataset that starts with the corresponding fields. You can use either the *IndexName* or the *IndexFieldNames* property.

Once you specify the index to use, use the *MasterFields* property to indicate the column(s) in the master dataset that correspond to the index fields in the detail table. To link datasets on multiple column names, separate field names with semicolons:

```
Parts.MasterFields := 'OrderNo;ItemNo';
```

To help create meaningful links between two datasets, you can use the Field Link designer. To use the Field Link designer, double click on the *MasterFields* property in the Object Inspector after you have assigned a *MasterSource* and an index.

The following steps create a simple form in which a user can scroll through customer records and display all orders for the current customer. The master table is the *CustomersTable* table, and the detail table is *OrdersTable*. The example uses the BDE-based *TTable* component, but you can use the same methods to link any table type datasets.

1 Place two *TTable* components and two *TDataSource* components in a data module.

2 Set the properties of the first *TTable* component as follows:

- *DatabaseName*: DBDEMOS
- *TableName*: CUSTOMER
- *Name*: CustomersTable

3 Set the properties of the second *TTable* component as follows:

- *DatabaseName*: DBDEMOS
- *TableName*: ORDERS
- *Name*: OrdersTable

4 Set the properties of the first *TDataSource* component as follows:

- *Name*: CustSource
- *DataSet*: CustomersTable

5 Set the properties of the second *TDataSource* component as follows:

- *Name*: OrdersSource
- *DataSet*: OrdersTable

6 Place two *TDBGrid* components on a form.

7 Choose File|Use Unit to specify that the form should use the data module.

8 Set the *DataSource* property of the first grid component to "CustSource", and set the *DataSource* property of the second grid to "OrdersSource".

9 Set the *MasterSource* property of *OrdersTable* to “CustSource”. This links the CUSTOMER table (the master table) to the ORDERS table (the detail table).

10 Double-click the *MasterFields* property value box in the Object Inspector to invoke the Field Link Designer to set the following properties:

- In the Available Indexes field, choose *CustNo* to link the two tables by the *CustNo* field.
- Select *CustNo* in both the Detail Fields and Master Fields field lists.
- Click the Add button to add this join condition. In the Joined Fields list, “CustNo -> CustNo” appears.
- Choose OK to commit your selections and exit the Field Link Designer.

11 Set the *Active* properties of *CustomersTable* and *OrdersTable* to *True* to display data in the grids on the form.

12 Compile and run the application.

If you run the application now, you will see that the tables are linked together, and that when you move to a new record in the CUSTOMER table, you see only those records in the ORDERS table that belong to the current customer.

### Using nested detail tables

A nested table is a detail dataset that is the value of a single dataset field in another (master) dataset. For datasets that represent server data, a nested detail dataset can only be used for a dataset field on the server. *TClientDataSet* components do not represent server data, but they can also contain dataset fields if you create a dataset for them that contains nested details, or if they receive data from a provider that is linked to the master table of a master/detail relationship.

**Note** For *TClientDataSet*, using nested detail sets is necessary if you want to apply updates from master and detail tables to a database server.

To use nested detail sets, the *ObjectView* property of the master dataset must be *True*. When your table type dataset contains nested detail datasets, *TDBGrid* provides support for displaying the nested details in a popup window. For more information on how this works, see “Displaying dataset fields” on page 25-27.

Alternately, you can display and edit detail datasets in data-aware controls by using a separate dataset component for the detail set. At design time, create persistent fields for the fields in your (master) dataset, using the Fields Editor: right click the master dataset and choose Fields Editor. Add a new persistent field to your dataset by right-clicking and choosing Add Fields. Define your new field with type DataSet Field. In the Fields Editor, define the structure of the detail table. You must also add persistent fields for any other fields used in your master dataset.

The dataset component for the detail table is a dataset descendant of a type allowed by the master table. *TTable* components only allow *TNestedDataSet* components as nested datasets. *TSQLTable* components allow other *TSQLTable* components. *TClientDataset* components allow other client datasets. Choose a dataset of the appropriate type from the Component palette and add it to your form or data module. Set this detail dataset's *DataSetField* property to the persistent DataSet field in the master dataset. Finally, place a data source component on the form or data module and set its *DataSet* property to the detail dataset. Data-aware controls can use this data source to access the data in the detail set.

## Controlling Read/write access to tables

By default when a table type dataset is opened, it requests read and write access for the underlying database table. Depending on the characteristics of the underlying database table, the requested write privilege may not be granted (for example, when you request write access to an SQL table on a remote server and the server restricts the table's access to read only).

**Note** This is not true for *TClientDataSet*, which determines whether users can edit data from information that the dataset provider supplies with data packets. It is also not true for *TSQLTable*, which is a unidirectional dataset, and hence always read-only.

When the table opens, you can check the *CanModify* property to ascertain whether the underlying database (or the dataset provider) allows users to edit the data in the table. If *CanModify* is *False*, the application cannot write to the database. If *CanModify* is *True*, your application can write to the database provided the table's *ReadOnly* property is *False*.

*ReadOnly* determines whether a user can both view and edit data. When *ReadOnly* is *False* (the default), a user can both view and edit data. To restrict a user to viewing data, set *ReadOnly* to *True* before opening the table.

**Note** *ReadOnly* is implemented on all table type datasets except *TSQLTable*, which is always read-only.

## Creating and deleting tables

Some table type datasets let you create and delete the underlying tables at design time or at runtime. Typically, database tables are created and deleted by a database administrator. However, it can be handy during application development and testing to create and destroy database tables that your application can use.

### Creating tables

*TTable* and *TIBTable* both let you create the underlying database table without using SQL. Similarly, *TClientDataSet* lets you create a dataset when you are not working with a dataset provider. Using *TTable* and *TClientDataSet*, you can create the table at design time or runtime. *TIBTable* only lets you create tables at runtime.

Before you can create the table, you must be set properties to specify the structure of the table you are creating. In particular, you must specify

- The database that will host the new table. For *TTable*, you specify the database using the *DatabaseName* property. For *TIBTable*, you must use a *TIBDatabase* component, which is assigned to the *Database* property. (Client datasets do not use a database.)
- The type of database (*TTable* only). Set the *TableType* property to the desired type of table. For Paradox, dBASE, or ASCII tables, set *TableType* to *ttParadox*, *ttDBase*, or *ttASCII*, respectively. For all other table types, set *TableType* to *ttDefault*.
- The name of the table you want to create. Both *TTable* and *TIBTable* have a *TableName* property for the name of the new table. Client datasets do not use a table name, but you should specify the *FileName* property before you save the new table. If you create a table that duplicates the name of an existing table, the existing table and all its data are overwritten by the newly created table. The old table and its data cannot be recovered. To avoid overwriting an existing table, you can check the *Exists* property at runtime. *Exists* is only available on *TTable* and *TIBTable*.
- The fields for the new table. There are two ways to do this:
  - You can add field definitions to the *FieldDefs* property. At design time, double-click the *FieldDefs* property in the Object Inspector to bring up the collection editor. Use the collection editor to add, remove, or change the properties of the field definitions. At runtime, clear any existing field definitions and then use the *AddFieldDef* method to add each new field definition. For each new field definition, set the properties of the *TFieldDef* object to specify the desired attributes of the field.
  - You can use persistent field components instead. At design time, double-click on the dataset to bring up the *Fields* editor. In the Fields editor, right-click and choose the New Field command. Describe the basic properties of your field. Once the field is created, you can alter its properties in the Object Inspector by selecting the field in the Fields editor.
- Indexes for the new table (optional). At design time, double-click the *IndexDefs* property in the Object Inspector to bring up the collection editor. Use the collection editor to add, remove, or change the properties of index definitions. At runtime, clear any existing index definitions, and then use the *AddIndexDef* method to add each new index definition. For each new index definition, set the properties of the *TIndexDef* object to specify the desired attributes of the index.

**Note** You can't define indexes for the new table if you are using persistent field components instead of field definition objects.

To create the table at design time, right-click the dataset and choose Create Table (*TTable*) or Create Data Set (*TClientDataSet*). This command does not appear on the context menu until you have specified all the necessary information.

To create the table at runtime, call the *CreateTable* method (*TTable* and *TIBTable*) or the *CreateDataSet* method (*TClientDataSet*).

**Note** You can set up the definitions at design time and then call the *CreateTable* (or *CreateDataSet*) method at runtime to create the table. However, to do so you must indicate that the definitions specified at runtime should be saved with the dataset component. (by default, field and index definitions are generated dynamically at runtime). Specify that the definitions should be saved with the dataset by setting its *StoreDefs* property to *True*.

**Tip** If you are using *TTable*, you can preload the field definitions and index definitions of an existing table at design time. Set the *DatabaseName* and *TableName* properties to specify the existing table. Right click the table component and choose Update Table Definition. This automatically sets the values of the *FieldDefs* and *IndexDefs* properties to describe the fields and indexes of the existing table. Next, reset the *DatabaseName* and *TableName* to specify the table you want to create, canceling any prompts to rename the existing table.

**Note** When creating Oracle8 tables, you can't create object fields (ADT fields, array fields, and dataset fields).

The following code creates a new table at runtime and associates it with the DBDEMOS alias. Before it creates the new table, it verifies that the table name provided does not match the name of an existing table:

```
var
  TableFound: Boolean;
begin
  with TTable.Create(nil) do // create a temporary TTable component
  begin
    try
      { set properties of the temporary TTable component }
      Active := False;
      DatabaseName := 'DBDEMOS';
      TableName := Edit1.Text;
      TableType := ttDefault;
      { define fields for the new table }
      FieldDefs.Clear;
      with FieldDefs.AddFieldDef do begin
        Name := 'First';
        DataType := ftString;
        Size := 20;
        Required := False;
      end;
      with FieldDefs.AddFieldDef do begin
        Name := 'Second';
        DataType := ftString;
        Size := 30;
        Required := False;
      end;
      { define indexes for the new table }
      IndexDefs.Clear;
      with IndexDefs.AddIndexDef do begin
        Name := '';
        Fields := 'First';
        Options := [ixPrimary];
      end;
      TableFound := Exists; // check whether the table already exists
      if TableFound then
        if MessageDlg('Overwrite existing table ' + Edit1.Text + '?',
             mtConfirmation, mbYesNoCancel, 0) = mrYes then
          TableFound := False;
      if not TableFound then
        CreateTable; // create the table
    finally
      Free; // destroy the temporary TTable when done
    end;
  end;
end;
```

### Deleting tables

*TTable* and *TIBTable* let you delete tables from the underlying database table without using SQL. To delete a table at runtime, call the dataset's *DeleteTable* method. For example, the following statement removes the table underlying a dataset:

```
CustomersTable.DeleteTable;
```

**Caution** When you delete a table with *DeleteTable*, the table and all its data are gone forever.

If you are using *TTable*, you can also delete tables at design time: Right-click the table component and select Delete Table from the context menu. The Delete Table menu pick is only present if the table component represents an existing database table (the *DatabaseName* and *TableName* properties specify an existing table).

## Emptying tables

Many table type datasets supply a single method that lets you delete all rows of data in the table.

- For *TTable* and *TIBTable*, you can delete all the records by calling the *EmptyTable* method at runtime:

  ```
  PhoneTable.EmptyTable;
  ```

- For *TADOTable*, you can use the *DeleteRecords* method.

  ```
  PhoneTable.DeleteRecords;
  ```

- For *TSQLTable*, you can use the *DeleteRecords* method as well. Note, however, that the *TSQLTable* version of *DeleteRecords* never takes any parameters.

  ```
  PhoneTable.DeleteRecords;
  ```

- For client datasets, you can use the *EmptyDataSet* method.

  ```
  PhoneTable.EmptyDataSet;
  ```

**Note** For tables on SQL servers, these methods only succeed if you have DELETE privilege for the table.

**Caution** When you empty a dataset, the data you delete is gone forever.

## Synchronizing tables

If you have two or more datasets that represent the same database table but do not share a data source component, then each dataset has its own view on the data and its own current record. As users access records through each datasets, the components' current records will differ.

If the datasets are all instances of *TTable*, or all instances of *TIBTable*, or all client datasets, you can force the current record for each of these datasets to be the same by calling the *GotoCurrent* method. *GotoCurrent* sets its own dataset's current record to the current record of a matching dataset. For example, the following code sets the current record of *CustomerTableOne* to be the same as the current record of *CustomerTableTwo*:

```
CustomerTableOne.GotoCurrent(CustomerTableTwo);
```

**Tip** If your application needs to synchronize datasets in this manner, put the datasets in a data module and add the unit for the data module to the uses clause of each unit that accesses the tables.

To synchronize datasets from separate forms, you must add one form's unit to the uses clause of the other, and you must qualify at least one of the dataset names with its form name. For example:

```
CustomerTableOne.GotoCurrent(Form2.CustomerTableTwo);
```

## Using query-type datasets

To use a query-type dataset,

1. Place the appropriate dataset component in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Identify the database server to query. Each query-type dataset does this differently, but typically you specify a database connection component:
   - For *TQuery*, specify a *TDatabase* component or a BDE alias using the *DatabaseName* property.
   - For *TADOQuery*, specify a *TADOConnection* component using the *Connection* property.
   - For *TSQLQuery*, specify a *TSQLConnection* component using the *SQLConnection* property.
   - For *TIBQuery*, specify a *TIBConnection* component using the *Database* property.

   For information about using database connection components, see Chapter 23, "Connecting to databases."
3. Specify an SQL statement in the *SQL* property of the dataset, and optionally specify any parameters for the statement. For more information, see "Specifying the query" on page 24-43 and "Using parameters in queries" on page 24-45.

4 If the query data is to be used with visual data controls, add a data source component to the data module, and set its *DataSet* property to the query-type dataset. The data source component forwards the results of the query (called a *result set*) to data-aware components for display. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.

5 Activate the query component. For queries that return a result set, use the *Active* property or the *Open* method. To execute queries that only perform an action on a table and return no result set, use the *ExecSQL* method at runtime. If you plan to execute the query more than once, you may want to call *Prepare* to initialize the data access layer and bind parameter values into the query. For information about preparing a query, see "Preparing queries" on page 24-48.

## Specifying the query

For true query-type datasets, you use the *SQL* property to specify the SQL statement for the dataset to execute. Some datasets, such as *TADODataSet*, *TSQLDataSet*, and client datasets, use a *CommandText* property to accomplish the same thing.

Most queries that return records are SELECT commands. Typically, they define the fields to include, the tables from which to select those fields, conditions that limit what records to include, and the order of the resulting dataset. For example:

```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = 1225
ORDER BY SaleDate
```

Queries that do not return records include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language.

The SQL command you execute must be acceptable to the server you are using. Datasets neither evaluate the SQL nor execute it. They merely pass the command to the server for execution. In most cases, the SQL command must be only one complete SQL statement, although that statement can be as complex as necessary (for example, a SELECT statement with a WHERE clause that uses several nested logical operators such as AND and OR). Some servers also support "batch" syntax that permits multiple statements; if your server supports such syntax, you can enter multiple statements when you specify the query.

The SQL statements used by queries can be verbatim, or they can contain replaceable parameters. Queries that use parameters are called *parameterized queries*. When you use parameterized queries, the actual values assigned to the parameters are inserted into the query before you execute, or run, the query. Using parameterized queries is very flexible, because you can change a user's view of and access to data on the fly at runtime without having to alter the SQL statement. For more information about parameterized queries, see "Using parameters in queries" on page 24-45.

### Specifying a query using the SQL property

When using a true query-type dataset *(TQuery*, *TADOQuery*, *TSQLQuery*, or *TIBQuery)*, assign the query to the *SQL* property. The *SQL* property is a *TStrings* object. Each separate string in this *TStrings* object is a separate line of the query. Using multiple lines does not affect the way the query executes on the server, but can make it easier to modify and debug the query if you divide the statement into logical units:

```
MyQuery.Close;
MyQuery.SQL.Clear;
MyQuery.SQL.Add('SELECT  CustNo, OrderNO,  SaleDate');
MyQuery.SQL.Add(' FROM Orders');
MyQuery.SQL.Add('ORDER  BY  SaleDate');
MyQuery.Open;
```

The code below demonstrates modifying only a single line in an existing SQL statement. In this case, the ORDER BY clause already exists on the third line of the statement. It is referenced via the *SQL* property using an index of 2.

```
MyQuery.SQL[2]  :=  ‘ORDER  BY  OrderNo’;
```

**Note** The dataset must be closed when you specify or modify the SQL property.

At design time, use the String List editor to specify the query. Click the ellipsis button by the *SQL* property in the Object Inspector to display the String List editor.

**Note** With some versions of Delphi, if you are using *TQuery*, you can also use the SQL Builder to construct a query based on a visible representation of tables and fields in a database. To use the SQL Builder, select the query component, right-click it to invoke the context menu, and choose Graphical Query Editor. To learn how to use SQL Builder, open it and use its online help.

Because the *SQL* property is a *TStrings* object, you can load the text of the query from a file by calling the *TStrings*.*LoadFromFile* method:

```
MyQuery.SQL.LoadFromFile('custquery.sql');
```

You can also use the *Assign* method of the *SQL* property to copy the contents of a string list object into the *SQL* property. The *Assign* method automatically clears the current contents of the *SQL* property before copying the new statement:

```
MyQuery.SQL.Assign(Memo1.Lines);
```

### Specifying a query using the CommandText property

When using *TADODataSet*, *TSQLDataSet,* or a client dataset, assign the text of the query statement to the *CommandText* property:

```
MyQuery.CommandText  :=  'SELECT  CustName,  Address FROM  Customer';
```

At design time, you can type the query directly into the Object Inspector, or, if the dataset already has an active connection to the database, you can click the ellipsis button by the *CommandText* property to display the Command Text editor. The Command Text editor lists the available tables, and the fields in those tables, to make it easier to compose your queries.

## Using parameters in queries

A parameterized SQL statement contains parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values, such as those used in a WHERE clause for comparisons, that appear in an SQL statement. Ordinarily, parameters stand in for data values passed to the statement. For example, in the following INSERT statement, values to insert are passed as parameters:

```
INSERT INTO Country (Name, Capital, Population)
VALUES (:Name, :Capital, :Population)
```

In this SQL statement, *:Name*, *:Capital*, and *:Population* are placeholders for actual values supplied to the statement at runtime by your application. Note that the names of parameters begin with a colon. The colon is required so that the parameter names can be distinguished from literal values. You can also include unnamed parameters by adding a question mark (?) to your query. Unnamed parameters are identified by position, because they do not have unique names.

Before the dataset can execute the query, you must supply values for any parameters in the query text. *TQuery*, *TIBQuery*, *TSQLQuery*, and client datasets use the *Params* property to store these values. *TADOQuery* uses the *Parameters* property instead. *Params* (or *Parameters*) is a collection of parameter objects (*TParam* or *TParameter*), where each object represents a single parameter. When you specify the text for the query, the dataset generates this set of parameter objects, and (depending on the dataset type) initializes any of their properties that it can deduce from the query.

**Note** You can suppress the automatic generation of parameter objects in response to changing the query text by setting the *ParamCheck* property to *False*. This is useful for data definition language (DDL) statements that contain parameters as part of the DDL statement that are not parameters for the query itself. For example, the DDL statement to create a stored procedure may define parameters that are part of the stored procedure. By setting *ParamCheck* to *False*, you prevent these parameters from being mistaken for parameters of the query.

Parameter values must be bound into the SQL statement before it is executed for the first time. Query components do this automatically for you even if you do not explicitly call the *Prepare* method before executing a query.

**Tip** It is a good programming practice to provide variable names for parameters that correspond to the actual name of the column with which it is associated. For example, if a column name is "Number," then its corresponding parameter would be ":Number". Using matching names is especially important if the dataset uses a data source to obtain parameter values from another dataset. This process is described in "Establishing master/detail relationships using parameters" on page 24-47.

### Supplying parameters at design time

At design time, you can specify parameter values using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* or *Parameters* property in the Object Inspector. If the SQL statement does not contain any parameters, no objects are listed in the collection editor.

**Note** The parameter collection editor is the same collection editor that appears for other collection properties. Because the editor is shared with other properties, its right-click context menu contains the Add and Delete commands. However, they are never enabled for query parameters. The only way to add or delete parameters is in the SQL statement itself.

For each parameter, select it in the parameter collection editor. Then use the Object Inspector to modify its properties.

When using the *Params* property (*TParam* objects), you will want to inspect or modify the following:

- The *DataType* property lists the data type for the parameter's value. For some datasets, this value may be correctly initialized. If the dataset did not deduce the type, *DataType* is *ftUnknown*, and you must change it to indicate the type of the parameter value.

  The *DataType* property lists the logical data type for the parameter. In general, these data types conform to server data types. For specific logical type-to-server data type mappings, see the documentation for the data access mechanism (BDE, dbExpress, InterBase).
- The *ParamType* property lists the type of the selected parameter. For queries, this is always *ptInput*, because queries can only contain input parameters. If the value of *ParamType* is *ptUnknown*, change it to *ptInput*.
- The *Value* property specifies a value for the selected parameter. You can leave *Value* blank if your application supplies parameter values at runtime.

When using the *Parameters* property (*TParameter* objects), you will want to inspect or modify the following:

- The *DataType* property lists the data type for the parameter's value. For some data types, you must provide additional information:
  - The *NumericScale* property indicates the number of decimal places for numeric parameters.
  - The *Precision* property indicates the total number of digits for numeric parameters.
  - The *Size* property indicates the number of characters in string parameters.
- The *Direction* property lists the type of the selected parameter. For queries, this is always *pdInput*, because queries can only contain input parameters.
- The *Attributes* property controls the type of values the parameter will accept. *Attributes* may be set to a combination of *psSigned*, *psNullable*, and *psLong*.
- The *Value* property specifies a value for the selected parameter. You can leave *Value* blank if your application supplies parameter values at runtime.

### Supplying parameters at runtime

To create parameters at runtime, you can use the

- *ParamByName* method to assign values to a parameter based on its name (not available for *TADOQuery*)
- *Params* or *Parameters* property to assign values to a parameter based on the parameter's ordinal position within the SQL statement.
- *Params.ParamValues or Parameters.ParamValues* property to assign values to one or more parameters in a single command line, based on the name of each parameter set.

The following code uses *ParamByName* to assign the text of an edit box to the :Capital parameter:

```
SQLQuery1.ParamByName('Capital').AsString := Edit1.Text;
```

The same code can be rewritten using the *Params* property, using an index of 0 (assuming the :Capital parameter is the first parameter in the SQL statement):

```
SQLQuery1.Params[0].AsString := Edit1.Text;
```

The command line below sets three parameters at once, using the *Params.ParamValues* property:

```
Query1.Params.ParamValues['Name;Capital;Continent'] :=
  VarArrayOf([Edit1.Text, Edit2.Text, Edit3.Text]);
```

Note that *ParamValues* uses Variants, avoiding the need to cast values.

## Establishing master/detail relationships using parameters

To set up a master/detail relationship where the detail set is a query-type dataset, you must specify a query that uses parameters. These parameters refer to current field values on the master dataset. Because the current field values on the master dataset change dynamically at runtime, you must rebind the detail set's parameters every time the master record changes. Although you could write code to do this using an event handler, all query-type datasets except *TIBQuery* provide an easier mechanism using the *DataSource* property.

If parameter values for a parameterized query are not bound at design time or specified at runtime, query-type datasets attempt to supply values for them based on the *DataSource* property. *DataSource* identifies a different dataset that is searched for field names that match the names of unbound parameters. This search dataset can be any type of dataset. The search dataset must be created and populated before you create the detail dataset that uses it. If matches are found in the search dataset, the detail dataset binds the parameter values to the values of the fields in the current record pointed to by the data source.

To illustrate how this works, consider two tables: a customer table and an orders table. For every customer, the orders table contains a set of orders that the customer made. The Customer table includes an ID field that specifies a unique customer ID. The orders table includes a CustID field that specifies the ID of the customer who made an order.

The first step is to set up the Customer dataset:

1 Add a table type dataset to your application and bind it to the Customer table.
2 Add a *TDataSource* component named *CustomerSource*. Set its *DataSet* property to the dataset added in step 1. This data source now represents the Customer dataset.
3 Add a query-type dataset and set its *SQL* property to

```
SELECT CustID, OrderNo, SaleDate
FROM Orders
WHERE CustID = :ID
```

   Note that the name of the parameter is the same as the name of the field in the master (Customer) table.
4 Set the detail dataset's *DataSource* property to *CustomerSource*. Setting this property makes the detail dataset a linked query.

At runtime the *:ID* parameter in the SQL statement for the detail dataset is not assigned a value, so the dataset tries to match the parameter by name against a column in the dataset identified by *CustomersSource*. *CustomersSource* gets its data from the master dataset, which, in turn, derives its data from the Customer table. Because the Customer table contains a column called "ID," the value from the *ID* field in the current record of the master dataset is assigned to the *:ID* parameter for the detail dataset's SQL statement. The datasets are linked in a master-detail relationship. Each time the current record changes in the Customers dataset, the detail dataset's SELECT statement executes to retrieve all orders based on the current customer id.

## Preparing queries

Preparing a query is an optional step that precedes query execution. Preparing a query submits the SQL statement and its parameters, if any, to the data access layer and the database server for parsing, resource allocation, and optimization. In some datasets, the dataset may perform additional setup operations when preparing the query. These operations improve query performance, making your application faster, especially when working with updatable queries.

An application can prepare a query by setting the *Prepared* property to *True*. If you do not prepare a query before executing it, the dataset automatically prepares it for you each time you call *Open* or *ExecSQL*. Even though the dataset prepares queries for you, you can improve performance by explicitly preparing the dataset before you open it the first time.

```
CustQuery.Prepared := True;
```

When you explicitly prepare the dataset, the resources allocated for executing the statement are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you add a parameter).

**Note** When you change the text of the *SQL* property for a query, the dataset automatically closes and unprepares the query.

## Executing queries that don't return a result set

When a query returns a set of records (such as a SELECT query), you execute the query the same way you populate any dataset with records: by setting *Active* to *True* or calling the *Open* method.

However, often SQL commands do not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records).

For all query-type datasets, you can execute a query that does not return a result set by calling *ExecSQL*:

```
CustomerQuery.ExecSQL; { query does not return a result set }
```

**Tip** If you are executing the query multiple times, it is a good idea to set the *Prepared* property to *True*.

Although the query does not return any records, you may want to know the number of records it affected (for example, the number of records deleted by a DELETE query). The *RowsAffected* property gives the number of affected records after a call to *ExecSQL*.

**Tip** When you do not know at design time whether the query returns a result set (for example, if the user supplies the query dynamically at runtime), you can code both types of query execution statements in a **try...except** block. Put a call to the *Open* method in the **try** clause. An action query is executed when the query is activated with the *Open* method, but an exception occurs in addition to that. Check the exception, and suppress it if it merely indicates the lack of a result set. (For example, *TQuery* indicates this by an *ENoResultSet* exception.)

## Using unidirectional result sets

When a query-type dataset returns a result set, it also receives a cursor, or pointer to the first record in that result set. The record pointed to by the cursor is the currently active record. The current record is the one whose field values are displayed in data-aware components associated with the result set's data source. Unless you are using dbExpress, this cursor is bi-directional by default. A bi-directional cursor can navigate both forward and backward through its records. Bi-directional cursor support requires some additional processing overhead, and can slow some queries.

If you do not need to be able to navigate backward through a result set, *TQuery* and *TIBQuery* let you improve query performance by requesting a unidirectional cursor instead. To request a unidirectional cursor, set the *UniDirectional* property to *True*.

Set *UniDirectional* before preparing and executing a query. The following code illustrates setting *UniDirectional* prior to preparing and executing a query:

```
if not (CustomerQuery.Prepared) then
begin
  CustomerQuery.UniDirectional := True;
  CustomerQuery.Prepared := True;
end;
CustomerQuery.Open;  { returns a result set with a one-way cursor }
```

**Note** Do not confuse the *UniDirectional* property with a unidirectional dataset. Unidirectional datasets (*TSQLDataSet*, *TSQLTable*, *TSQLQuery*, and *TSQLStoredProc*) use dbExpress, which only returns unidirectional cursors. In addition to restricting the ability to navigate backwards, unidirectional datasets do not buffer records, and so have additional limitations (such as the inability to use filters).

## Using stored procedure-type datasets

How your application uses a stored procedure depends on how the stored procedure was coded, whether and how it returns data, the specific database server used, or a combination of these factors.

In general terms, to access a stored procedure on a server, an application must:

1. Place the appropriate dataset component in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Identify the database server that defines the stored procedure. Each stored procedure-type dataset does this differently, but typically you specify a database connection component:
   - For *TStoredProc*, specify a *TDatabase* component or a BDE alias using the *DatabaseName* property.
   - For *TADOStoredProc*, specify a *TADOConnection* component using the *Connection* property.
   - For *TSQLStoredProc*, specify a *TSQLConnection* component using the *SQLConnection* property.
   - For *TIBStoredProc*, specify a *TIBConnection* component using the *Database* property.

   For information about using database connection components, see Chapter 23, “Connecting to databases.”
3. Specify the stored procedure to execute. For most stored procedure-type datasets, you do this by setting the *StoredProcName* property. The one exception is *TADOStoredProc*, which has a *ProcedureName* property instead.

4 If the stored procedure returns a cursor to be used with visual data controls, add a data source component to the data module, and set its *DataSet* property to the stored procedure-type dataset. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.

5 Provide input parameter values for the stored procedure, if necessary. If the server does not provide information about all stored procedure parameters, you may need to provide additional input parameter information, such as parameter names and data types. For information about working with stored procedure parameters, see "Working with stored procedure parameters" on page 24-51.

6 Execute the stored procedure. For stored procedures that return a cursor, use the *Active* property or the *Open* method. To execute stored procedures that do not return any results or that only return output parameters, use the *ExecProc* method at runtime. If you plan to execute the stored procedure more than once, you may want to call *Prepare* to initialize the data access layer and bind parameter values into the stored procedure. For information about preparing a query, see "Executing stored procedures that don't return a result set" on page 24-55.

7 Process any results. These results can be returned as result and output parameters, or they can be returned as a result set that populates the stored procedure-type dataset. Some stored procedures return multiple cursors. For details on how to access the additional cursors, see "Fetching multiple result sets" on page 24-56.

## Working with stored procedure parameters

There are four types of parameters that can be associated with stored procedures:

- *Input parameters*, used to pass values to a stored procedure for processing.
- *Output parameters*, used by a stored procedure to pass return values to an application.
- *Input/output parameters*, used to pass values to a stored procedure for processing, and used by the stored procedure to pass return values to the application.
- A *result parameter*, used by some stored procedures to return an error or status value to an application. A stored procedure can only return one result parameter.

Whether a stored procedure uses a particular type of parameter depends both on the general language implementation of stored procedures on your database server and on a specific instance of a stored procedure. For any server, individual stored procedures may or may not use input parameters. On the other hand, some uses of parameters are server-specific. For example, on MS-SQL Server and Sybase stored procedures always return a result parameter, but the InterBase implementation of a stored procedure never returns a result parameter.

Access to stored procedure parameters is provided by the *Params* property (in *TStoredProc*, *TSQLStoredProc*, *TIBStoredProc*) or the *Parameters* property (in *TADOStoredProc*). When you assign a value to the *StoredProcName* (or *ProcedureName*) property, the dataset automatically generates objects for each parameter of the stored procedure. For some datasets, if the stored procedure name is not specified until runtime, objects for each parameter must be programmatically created at that time. Not specifying the stored procedure and manually creating the *TParam* or *TParameter* objects allows a single dataset to be used with any number of available stored procedures.

**Note** Some stored procedures return a dataset in addition to output and result parameters. Applications can display dataset records in data-aware controls, but must separately process output and result parameters.

## Setting up parameters at design time

You can specify stored procedure parameter values at design time using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* or *Parameters* property in the Object Inspector.

**Important** You can assign values to input parameters by selecting them in the parameter collection editor and using the Object Inspector to set the *Value* property. However, do not change the names or data types for input parameters reported by the server. Otherwise, when you execute the stored procedure an exception is raised.

Some servers do not report parameter names or data types. In these cases, you must set up the parameters manually using the parameter collection editor. Right click and choose Add to add parameters. For each parameter you add, you must fully describe the parameter. Even if you do not need to add any parameters, you should check the properties of individual parameter objects to ensure that they are correct.

If the dataset has a *Params* property (*TParam* objects), the following properties must be correctly specified:

- The *Name* property indicates the name of the parameter as it is defined by the stored procedure.
- The *DataType* property gives the data type for the parameter's value. When using *TSQLStoredProc*, some data types require additional information:
  - The *NumericScale* property indicates the number of decimal places for numeric parameters.
  - The *Precision* property indicates the total number of digits for numeric parameters.
  - The *Size* property indicates the number of characters in string parameters.

- The *ParamType* property indicates the type of the selected parameter. This can be *ptInput* (for input parameters), *ptOutput* (for output parameters), *ptInputOutput* (for input/output parameters) or *ptResult* (for result parameters).
- The *Value* property specifies a value for the selected parameter. You can never set values for output and result parameters. These types of parameters have values set by the execution of the stored procedure. For input and input/output parameters, you can leave *Value* blank if your application supplies parameter values at runtime.

If the dataset uses a *Parameters* property (*TParameter* objects), the following properties must be correctly specified:

- The *Name* property indicates the name of the parameter as it is defined by the stored procedure.
- The *DataType* property gives the data type for the parameter's value. For some data types, you must provide additional information:
  - The *NumericScale* property indicates the number of decimal places for numeric parameters.
  - The *Precision* property indicates the total number of digits for numeric parameters.
  - The *Size* property indicates the number of characters in string parameters.
- The *Direction* property gives the type of the selected parameter. This can be *pdInput* (for input parameters), *pdOutput* (for output parameters), *pdInputOutput* (for input/output parameters) or *pdReturnValue* (for result parameters).
- The *Attributes* property controls the type of values the parameter will accept. *Attributes* may be set to a combination of *psSigned*, *psNullable*, and *psLong*.
- The *Value* property specifies a value for the selected parameter. Do not set values for output and result parameters. For input and input/output parameters, you can leave *Value* blank if your application supplies parameter values at runtime.

## Using parameters at runtime

With some datasets, if the name of the stored procedure is not specified until runtime, no *TParam* objects are automatically created for parameters and they must be created programmatically. This can be done using the *TParam.Create* method or the *TParams.AddParam* method:

```
var
  P1, P2: TParam;
begin
  ƒ
  with StoredProc1 do begin
    StoredProcName := 'GET_EMP_PROJ';
    Params.Clear;
    P1 := TParam.Create(Params, ptInput);
    P2 := TParam.Create(Params, ptOutput);
    try
      Params[0].Name := ‘EMP_NO’;
      Params[1].Name := ‘PROJ_ID’;
      ParamByname(‘EMP_NO’).AsSmallInt := 52;
      ExecProc;
      Edit1.Text := ParamByname(‘PROJ_ID’).AsString;
    finally
      P1.Free;
      P2.Free;
    end;
  end;
  ƒ
end;
```

Even if you do not need to add the individual parameter objects at runtime, you may want to access individual parameter objects to assign values to input parameters and to retrieve values from output parameters. You can use the dataset’s *ParamByName* method to access individual parameters based on their names. For example, the following code sets the value of an input/output parameter, executes the stored procedure, and retrieves the returned value:

```
with SQLStoredProc1 do
begin
  ParamByName('IN_OUTVAR').AsInteger := 103;
  ExecProc;
  IntegerVar := ParamByName('IN_OUTVAR').AsInteger;
end;
```

## Preparing stored procedures

As with query-type datasets, stored procedure-type datasets must be prepared before they execute the stored procedure. Preparing a stored procedure tells the data access layer and the database server to allocate resources for the stored procedure and to bind parameters. These operations can improve performance.

If you attempt to execute a stored procedure before preparing it, the dataset automatically prepares it for you, and then unprepares it after it executes. If you plan to execute a stored procedure a number of times, it is more efficient to explicitly prepare it by setting the Prepared property to *True*.

```
MyProc.Prepared := True;
```

When you explicitly prepare the dataset, the resources allocated for executing the stored procedure are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you change the parameters when using Oracle overloaded procedures).

## Executing stored procedures that don’t return a result set

When a stored procedure returns a cursor, you execute it the same way you populate any dataset with records: by setting *Active* to *True* or calling the *Open* method.

However, often stored procedures do not return any data, or only return results in output parameters. You can execute a stored procedure that does not return a result set by calling *ExecProc*. After executing the stored procedure, you can use the *ParamByName* method to read the value of the result parameter or of any output parameters:

```
MyStoredProcedure.ExecProc; { does not return a result set }
Edit1.Text := MyStoredProcedure.ParamByName('OUTVAR').AsString;
```

**Note** *TADOStoredProc* does not have a *ParamByName* method. To obtain output parameter values when using ADO, access parameter objects using the *Parameters* property.

**Tip** If you are executing the procedure multiple times, it is a good idea to set the *Prepared* property to *True*.

## Fetching multiple result sets

Some stored procedures return multiple sets of records. The dataset only fetches the first set when you open it. If you are using *TSQLStoredProc* or *TADOStoredProc*, you can access the other sets of records by calling the *NextRecordSet* method:

```
var
  DataSet2: TCustomSQLDataSet;
begin
  DataSet2 := SQLStoredProc1.NextRecordSet;
  ...
```

In *TSQLStoredProc, NextRecordSet* returns a newly created *TCustomSQLDataSet* component that provides access to the next set of records. In *TADOStoredProc*, *NextRecordset* returns an interface that can be assigned to the *RecordSet* property of an existing ADO dataset. For either class, the method returns the number of records in the returned dataset as an output parameter.

The first time you call *NextRecordSet*, it returns the second set of records. Calling *NextRecordSet* again returns a third dataset, and so on, until there are no more sets of records. When there are no additional cursors, *NextRecordSet* returns **nil**.

Chapter

25

# Working with field components

This chapter describes the properties, events, and methods common to the *TField* object and its descendants. Field components represent individual fields (columns) in datasets. This chapter also describes how to use field components to control the display and editing of data in your applications.

Field components are always associated with a dataset. You never use a *TField* object directly in your applications. Instead, each field component in your application is a *TField* descendant specific to the datatype of a column in a dataset. Field components provide data-aware controls such as *TDBEdit* and *TDBGrid* access to the data in a particular column of the associated dataset.

Generally speaking, a single field component represents the characteristics of a single column, or field, in a dataset, such as its data type and size. It also represents the field's display characteristics, such as alignment, display format, and edit format. For example, a *TFloatField* component has four properties that directly affect the appearance of its data:

**Table 25.1** TFloatField properties that affect data display

| Property | Purpose |
|---|---|
| Alignment | Specifies whether data is displayed left-aligned, centered, or right-aligned. |
| DisplayWidth | Specifies the number of digits to display in a control at one time. |
| DisplayFormat | Specifies data formatting for display (such as how many decimal places to show). |
| EditFormat | Specifies how to display a value during editing. |

As you scroll from record to record in a dataset, a field component lets you view and change the value for that field in the current record.

Field components have many properties in common with one another (such as *DisplayWidth* and *Alignment*), and they have properties specific to their data types (such as *Precision* for *TFloatField*). Each of these properties affect how data appears to an application's users on a form. Some properties, such as *Precision*, can also affect what data values the user can enter in a control when modifying or entering data.

All field components for a dataset are either *dynamic* (automatically generated for you based on the underlying structure of database tables), or *persistent* (generated based on specific field names and properties you set in the Fields editor). Dynamic and persistent fields have different strengths and are appropriate for different types of applications. The following sections describe dynamic and persistent fields in more detail and offer advice on choosing between them.

## Dynamic field components

Dynamically generated field components are the default. In fact, all field components for any dataset start out as dynamic fields the first time you place a dataset on a data module, specify how that dataset fetches its data, and open it. A field component is *dynamic* if it is created automatically based on the underlying physical characteristics of the data represented by a dataset. Datasets generate one field component for each column in the underlying data. The exact *TField* descendant created for each column is determined by field type information received from the database or (for *TClientDataSet*) from a provider component.

Dynamic fields are temporary. They exist only as long as a dataset is open. Each time you reopen a dataset that uses dynamic fields, it rebuilds a completely new set of dynamic field components based on the current structure of the data underlying the dataset. If the columns in the underlying data change, then the next time you open a dataset that uses dynamic field components, the automatically generated field components are also changed to match.

Use dynamic fields in applications that must be flexible about data display and editing. For example, to create a database browsing tool such as SQL explorer, you must use dynamic fields because every database table has different numbers and types of columns. You might also want to use dynamic fields in applications where user interaction with data mostly takes place inside grid components and you know that the datasets used by the application change frequently.

To use dynamic fields in an application:

1. Place datasets and data sources in a data module.
2. Associate the datasets with data. This involves using a connection component or provider to connect to the source of the data and setting any properties that specify what data the dataset represents.
3. Associate the data sources with the datasets.

4 Place data-aware controls in the application's forms, add the data module to each uses clause for each form's unit, and associate each data-aware control with a data source in the module. In addition, associate a field with each data-aware control that requires one. Note that because you are using dynamic field components, there is no guarantee that any field name you specify will exist when the dataset is opened.

5 Open the datasets.

Aside from ease of use, dynamic fields can be limiting. Without writing code, you cannot change the display and editing defaults for dynamic fields, you cannot safely change the order in which dynamic fields are displayed, and you cannot prevent access to any fields in the dataset. You cannot create additional fields for the dataset, such as calculated fields or lookup fields, and you cannot override a dynamic field's default data type. To gain control and flexibility over fields in your database applications, you need to invoke the Fields editor to create persistent field components for your datasets.

## Persistent field components

By default, dataset fields are dynamic. Their properties and availability are automatically set and cannot be changed in any way. To gain control over a field's properties and events you must create persistent fields for the dataset. Persistent fields let you

- Set or change the field's display or edit characteristics at design time or runtime.
- Create new fields, such as lookup fields, calculated fields, and aggregated fields, that base their values on existing fields in a dataset.
- Validate data entry.
- Remove field components from the list of persistent components to prevent your application from accessing particular columns in an underlying database.
- Define new fields to replace existing fields, based on columns in the table or query underlying a dataset.

At design time, you can—and should—use the Fields editor to create persistent lists of the field components used by the datasets in your application. Persistent field component lists are stored in your application, and do not change even if the structure of a database underlying a dataset is changed. Once you create persistent fields with the Fields editor, you can also create event handlers for them that respond to changes in data values and that validate data entries.

**Note** When you create persistent fields for a dataset, only those fields you select are available to your application at design time and runtime. At design time, you can always use the Fields editor to add or remove persistent fields for a dataset.

All fields used by a single dataset are either persistent or dynamic. You cannot mix field types in a single dataset. If you create persistent fields for a dataset, and then want to revert to dynamic fields, you must remove all persistent fields from the dataset. For more information about dynamic fields, see "Dynamic field components" on page 25-2.

**Note** One of the primary uses of persistent fields is to gain control over the appearance and display of data. You can also control the appearance of columns in data-aware grids. To learn about controlling column appearance in grids, see "Creating a customized grid" on page 20-17.

## Creating persistent fields

Persistent field components created with the Fields editor provide efficient, readable, and type-safe programmatic access to underlying data. Using persistent field components guarantees that each time your application runs, it always uses and displays the same columns, in the same order even if the physical structure of the underlying database has changed. Data-aware components and program code that rely on specific fields always work as expected. If a column on which a persistent field component is based is deleted or changed, Delphi generates an exception rather than running the application against a nonexistent column or mismatched data.

To create persistent fields for a dataset:

1. Place a dataset in a data module.
2. Bind the dataset to its underlying data. This typically involves associating the dataset with a connection component or provider and specifying any properties to describe the data. For example, If you are using *TADODataSet*, you can set the *Connection* property to a properly configured *TADOConnection* component and set the *CommandText* property to a valid query.
3. Double-click the dataset component in the data module to invoke the Fields editor. The Fields editor contains a title bar, navigator buttons, and a list box.

   The title bar of the Fields editor displays both the name of the data module or form containing the dataset, and the name of the dataset itself. For example, if you open the *Customers* dataset in the *CustomerData* data module, the title bar displays 'CustomerData.Customers,' or as much of the name as fits.

   Below the title bar is a set of navigation buttons that let you scroll one-by-one through the records in an active dataset at design time, and to jump to the first or last record. The navigation buttons are dimmed if the dataset is not active or if the dataset is empty. If the dataset is unidirectional, the buttons for moving to the last record and the previous record are always dimmed.

   The list box displays the names of persistent field components for the dataset. The first time you invoke the Fields editor for a new dataset, the list is empty because the field components for the dataset are dynamic, not persistent. If you invoke the Fields editor for a dataset that already has persistent field components, you see the field component names in the list box.

4 Choose Add Fields from the Fields editor context menu.

5 Select the fields to make persistent in the Add Fields dialog box. By default, all fields are selected when the dialog box opens. Any fields you select become persistent fields.

The Add Fields dialog box closes, and the fields you selected appear in the Fields editor list box. Fields in the Fields editor list box are persistent. If the dataset is active, note, too, that the Next and (if the dataset is not unidirectional) Last navigation buttons above the list box are enabled.

From now on, each time you open the dataset, it no longer creates dynamic field components for every column in the underlying database. Instead it only creates persistent components for the fields you specified.

Each time you open the dataset, it verifies that each non-calculated persistent field exists or can be created from data in the database. If it cannot, the dataset raises an exception warning you that the field is not valid, and does not open the dataset.

## Arranging persistent fields

The order in which persistent field components are listed in the Fields editor list box is the default order in which the fields appear in a data-aware grid component. You can change field order by dragging and dropping fields in the list box.

To change the order of fields:

1 Select the fields. You can select and order one or more fields at a time.

2 Drag the fields to a new location.

If you select a noncontiguous set of fields and drag them to a new location, they are inserted as a contiguous block. Within the block, the order of fields does not change.

Alternatively, you can select the field, and use Ctrl+Up and Ctrl+Dn to change an individual field's order in the list.

## Defining new persistent fields

Besides making existing dataset fields into persistent fields, you can also create special persistent fields as additions to or replacements of the other persistent fields in a dataset.

New persistent fields that you create are only for display purposes. The data they contain at runtime are not retained either because they already exist elsewhere in the database, or because they are temporary. The physical structure of the data underlying the dataset is not changed in any way.

To create a new persistent field component, invoke the context menu for the Fields editor and choose New field. The New Field dialog box appears.

The New Field dialog box contains three group boxes: Field properties, Field type, and Lookup definition.

- The Field properties group box lets you enter general field component information. Enter the field name in the Name edit box. The name you enter here corresponds to the field component's *FieldName* property. The New Field dialog uses this name to build a component name in the Component edit box. The name that appears in the Component edit box corresponds to the field component's *Name* property and is only provided for informational purposes (*Name* is the identifier by which you refer to the field component in your source code). The dialog discards anything you enter directly in the Component edit box.
- The Type combo box in the Field properties group lets you specify the field component's data type. You must supply a data type for any new field component you create. For example, to display floating-point currency values in a field, select *Currency* from the drop-down list. Use the Size edit box to specify the maximum number of characters that can be displayed or entered in a string-based field, or the size of *Bytes* and *VarBytes* fields. For all other data types, Size is meaningless.
- The Field type radio group lets you specify the type of new field component to create. The default type is Data. If you choose Lookup, the Dataset and Source Fields edit boxes in the Lookup definition group box are enabled. You can also create Calculated fields, and if you are working with a client dataset, you can create InternalCalc fields or Aggregate fields. The following table describes these types of fields you can create:

**Table 25.2** Special persistent field kinds

| Field kind | Purpose |
|---|---|
| Data | Replaces an existing field (for example to change its data type) |
| Calculated | Displays values calculated at runtime by a dataset's *OnCalcFields* event handler. |
| Lookup | Retrieve values from a specified dataset at runtime based on search criteria you specify. (not supported by unidirectional datasets) |
| InternalCalc | Displays values calculated at runtime by a client dataset and stored with its data. |
| Aggregate | Displays a value summarizing the data in a set of records from a client dataset. |

The Lookup definition group box is only used to create lookup fields. This is described more fully in "Defining a lookup field" on page 25-9.

## Defining a data field

A data field replaces an existing field in a dataset. For example, for programmatic reasons you might want to replace a *TSmallIntField* with a *TIntegerField*. Because you cannot change a field's data type directly, you must define a new field to replace it.

**Important** Even though you define a new field to replace an existing field, the field you define must derive its data values from an existing column in a table underlying a dataset.

To create a replacement data field for a field in a table underlying a dataset, follow these steps:

1 Remove the field from the list of persistent fields assigned for the dataset, and then choose New Field from the context menu.

2 In the New Field dialog box, enter the name of an existing field in the database table in the Name edit box. Do not enter a new field name. You are actually specifying the name of the field from which your new field will derive its data.

3 Choose a new data type for the field from the Type combo box. The data type you choose should be different from the data type of the field you are replacing. You cannot replace a string field of one size with a string field of another size. Note that while the data type should be different, it must be compatible with the actual data type of the field in the underlying table.

4 Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.

5 Select Data in the Field type radio group if it is not already selected.

6 Choose OK. The New Field dialog box closes, the newly defined data field replaces the existing field you specified in Step 1, and the component declaration in the data module or form's **type** declaration is updated.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector. For more information about editing field component properties and events, see "Setting persistent field properties and events" on page 25-11.

## Defining a calculated field

A calculated field displays values calculated at runtime by a dataset's *OnCalcFields* event handler. For example, you might create a string field that displays concatenated values from other fields.

To create a calculated field in the New Field dialog box:

1 Enter a name for the calculated field in the Name edit box. Do not enter the name of an existing field.

2 Choose a data type for the field from the Type combo box.

3 Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.

4 Select Calculated or InternalCalc in the Field type radio group. InternalCalc is only available if you are working with a client dataset. The significant difference between these types of calculated fields is that the values calculated for an InternalCalc field are stored and retrieved as part of the client dataset's data.

5 Choose OK. The newly defined calculated field is automatically added to the end of the list of persistent fields in the Field editor list box, and the component declaration is automatically added to the form's or data module's **type** declaration.

6 Place code that calculates values for the field in the *OnCalcFields* event handler for the dataset. For more information about writing code to calculate field values, see "Programming a calculated field" on page 25-8.

**Note** To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector. For more information about editing field component properties and events, see "Setting persistent field properties and events" on page 25-11.

## Programming a calculated field

After you define a calculated field, you must write code to calculate its value. Otherwise, it always has a null value. Code for a calculated field is placed in the *OnCalcFields* event for its dataset.

To program a value for a calculated field:

1 Select the dataset component from the Object Inspector drop-down list.

2 Choose the Object Inspector Events page.

3 Double-click the *OnCalcFields* property to bring up or create a *CalcFields* procedure for the dataset component.

4 Write the code that sets the values and other properties of the calculated field as desired.

For example, suppose you have created a *CityStateZip* calculated field for the *Customers* table on the *CustomerData* data module. *CityStateZip* should display a company's city, state, and zip code on a single line in a data-aware control.

To add code to the *CalcFields* procedure for the *Customers* table, select the *Customers* table from the Object Inspector drop-down list, switch to the Events page, and double-click the *OnCalcFields* property.

The *TCustomerData.CustomersCalcFields* procedure appears in the unit's source code window. Add the following code to the procedure to calculate the field:

```
CustomersCityStateZip.Value := CustomersCity.Value + ', ' + CustomersState.Value
  + ' ' + CustomersZip.Value;
```

**Note** When writing the *OnCalcFields* event handler for an internally calculated field, you can improve performance by checking the client dataset's *State* property and only recomputing the value when *State* is *dsInternalCalc*. See "Using internally calculated fields in client datasets" on page 29-11 for details.

### Defining a lookup field

A lookup field is a read-only field that displays values at runtime based on search criteria you specify. In its simplest form, a lookup field is passed the name of an existing field to search on, a field value to search for, and a different field in a lookup dataset whose value it should display.

For example, consider a mail-order application that enables an operator to use a lookup field to determine automatically the city and state that correspond to the zip code a customer provides. The column to search on might be called *ZipTable.Zip*, the value to search for is the customer's zip code as entered in *Order.CustZip*, and the values to return would be those for the *ZipTable.City* and *ZipTable.State* columns of the record where the value of *ZipTable.Zip* matches the current value in the *Order.CustZip* field.

**Note** Unidirectional datasets do not support lookup fields.

To create a lookup field in the New Field dialog box:

1. Enter a name for the lookup field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
4. Select Lookup in the Field type radio group. Selecting Lookup enables the Dataset and Key Fields combo boxes.
5. Choose from the Dataset combo box drop-down list the dataset in which to look up field values. The lookup dataset must be different from the dataset for the field component itself, or a circular reference exception is raised at runtime. Specifying a lookup dataset enables the Lookup Keys and Result Field combo boxes.
6. Choose from the Key Fields drop-down list a field in the current dataset for which to match values. To match more than one field, enter field names directly instead of choosing from the drop-down list. Separate multiple field names with semicolons. If you are using more than one field, you must use persistent field components.
7. Choose from the Lookup Keys drop-down list a field in the lookup dataset to match against the Source Fields field you specified in step 6. If you specified more than one key field, you must specify the same number of lookup keys. To specify more than one field, enter field names directly, separating multiple field names with semicolons.
8. Choose from the Result Field drop-down list a field in the lookup dataset to return as the value of the lookup field you are creating.

When you design and run your application, lookup field values are determined before calculated field values are calculated. You can base calculated fields on lookup fields, but you cannot base lookup fields on calculated fields.

You can use the *LookupCache* property to hone the way lookup fields are determined. *LookupCache* determines whether the values of a lookup field are cached in memory when a dataset is first opened, or looked up dynamically every time the current record in the dataset changes. Set *LookupCache* to *True* to cache the values of a lookup field when the *LookupDataSet* is unlikely to change and the number of distinct lookup values is small. Caching lookup values can speed performance, because the lookup values for every set of *LookupKeyFields* values are preloaded when the *DataSet* is opened. When the current record in the *DataSet* changes, the field object can locate its *Value* in the cache, rather than accessing the *LookupDataSet*. This performance improvement is especially dramatic if the *LookupDataSet* is on a network where access is slow.

**Tip** You can use a lookup cache to provide lookup values programmatically rather than from a secondary dataset. Be sure that the *LookupDataSet* property is nil. Then, use the *LookupList* property's *Add* method to fill it with lookup values. Set the *LookupCache* property to *True*. The field will use the supplied lookup list without overwriting it with values from a lookup dataset.

If every record of *DataSet* has different values for *KeyFields*, the overhead of locating values in the cache can be greater than any performance benefit provided by the cache. The overhead of locating values in the cache increases with the number of distinct values that can be taken by *KeyFields*.

If *LookupDataSet* is volatile, caching lookup values can lead to inaccurate results. Call *RefreshLookupList* to update the values in the lookup cache. *RefreshLookupList* regenerates the *LookupList* property, which contains the value of the *LookupResultField* for every set of *LookupKeyFields* values.

When setting *LookupCache* at runtime, call *RefreshLookupList* to initialize the cache.

### Defining an aggregate field

An aggregate field displays values from a maintained aggregate in a client dataset. An aggregate is a calculation that summarizes the data in a set of records. See "Using maintained aggregates" on page 29-11 for details about maintained aggregates.

To create an aggregate field in the New Field dialog box:

1. Enter a name for the aggregate field in the Name edit box. Do not enter the name of an existing field.
2. Choose aggregate data type for the field from the Type combo box.
3. Select Aggregate in the Field type radio group.
4. Choose OK. The newly defined aggregate field is automatically added to the client dataset and its *Aggregates* property is automatically updated to include the appropriate aggregate specification.
5. Place the calculation for the aggregate in the *ExprText* property of the newly created aggregate field. For more information about defining an aggregate, see "Specifying aggregates" on page 29-12.

Once a persistent *TAggregateField* is created, a *TDBText* control can be bound to the aggregate field. The *TDBText* control will then display the value of the aggregate field that is relevant to the current record of the underlying client data set.

## Deleting persistent field components

Deleting a persistent field component is useful for accessing a subset of available columns in a table, and for defining your own persistent fields to replace a column in a table. To remove one or more persistent field components for a dataset:

1 Select the field(s) to remove in the Fields editor list box.

2 Press Del.

**Note** You can also delete selected fields by invoking the context menu and choosing Delete.

Fields you remove are no longer available to the dataset and cannot be displayed by data-aware controls. You can always recreate a persistent field component that you delete by accident, but any changes previously made to its properties or events is lost. For more information, see “Creating persistent fields” on page 25-4.

**Note** If you remove all persistent field components for a dataset, the dataset reverts to using dynamic field components for every column in the underlying database table.

## Setting persistent field properties and events

You can set properties and customize events for persistent field components at design time. Properties control the way a field is displayed by a data-aware component, for example, whether it can appear in a *TDBGrid*, or whether its value can be modified. Events control what happens when data in a field is fetched, changed, set, or validated.

To set the properties of a field component or write customized event handlers for it, select the component in the Fields editor, or select it from the component list in the Object Inspector.

### Setting display and edit properties at design time

To edit the display properties of a selected field component, switch to the Properties page on the Object Inspector window. The following table summarizes display properties that can be edited.

**Table 25.3** Field component properties

| Property | Purpose |
|---|---|
| *Alignment* | Left justifies, right justifies, or centers a field contents within a data-aware component. |
| *ConstraintErrorMessage* | Specifies the text to display when edits clash with a constraint condition. |
| *CustomConstraint* | Specifies a local constraint to apply to data during editing. |

**Table 25.3** Field component properties (continued)

| Property | Purpose |
|---|---|
| *Currency* | Numeric fields only.<br>*True*: displays monetary values.<br>*False* (default): does not display monetary values. |
| *DisplayFormat* | Specifies the format of data displayed in a data-aware component. |
| *DisplayLabel* | Specifies the column name for a field in a data-aware grid component. |
| *DisplayWidth* | Specifies the width, in characters, of a grid column that display this field. |
| *EditFormat* | Specifies the edit format of data in a data-aware component. |
| *EditMask* | Limits data-entry in an editable field to specified types and ranges of characters, and specifies any special, non-editable characters that appear within the field (hyphens, parentheses, and so on). |
| *FieldKind* | Specifies the type of field to create. |
| *FieldName* | Specifies the actual name of a column in the table from which the field derives its value and data type. |
| *HasConstraints* | Indicates whether there are constraint conditions imposed on a field. |
| *ImportedConstraint* | Specifies an SQL constraint imported from the Data Dictionary or an SQL server. |
| *Index* | Specifies the order of the field in a dataset. |
| *LookupDataSet* | Specifies the table used to look up field values when *Lookup* is *True*. |
| *LookupKeyFields* | Specifies the field(s) in the lookup dataset to match when doing a lookup. |
| *LookupResultField* | Specifies the field in the lookup dataset from which to copy values into this field. |
| *MaxValue* | Numeric fields only. Specifies the maximum value a user can enter for the field. |
| *MinValue* | Numeric fields only. Specifies the minimum value a user can enter for the field. |
| *Name* | Specifies the component name of the field component within Delphi. |
| *Origin* | Specifies the name of the field as it appears in the underlying database. |
| *Precision* | Numeric fields only. Specifies the number of significant digits. |
| *ReadOnly* | *True*: Displays field values in data-aware controls, but prevents editing.<br>*False* (the default): Permits display and editing of field values. |
| *Size* | Specifies the maximum number of characters that can be displayed or entered in a string-based field, or the size, in bytes, of *TBytesField* and *TVarBytesField* fields. |
| *Tag* | Long integer bucket available for programmer use in every component as needed. |
| *Transliterate* | *True* (default): specifies that translation to and from the respective locales will occur as data is transferred between a dataset and a database.<br>*False*: Locale translation does not occur. |
| *Visible* | *True* (the default): Permits display of field in a data-aware grid.<br>*False*: Prevents display of field in a data-aware grid component.<br>User-defined components can make display decisions based on this property. |

Not all properties are available for all field components. For example, a field component of type *TStringField* does not have *Currency*, *MaxValue*, or *DisplayFormat* properties, and a component of type *TFloatField* does not have a *Size* property.

While the purpose of most properties is straightforward, some properties, such as *Calculated*, require additional programming steps to be useful. Others, such as *DisplayFormat*, *EditFormat*, and *EditMask*, are interrelated; their settings must be coordinated. For more information about using *DisplayFormat*, *EditFormat*, and *EditMask*, see "Controlling and masking user input" on page 25-15.

## Setting field component properties at runtime

You can use and manipulate the properties of field component at runtime. Access persistent field components by name, where the name can be obtained by concatenating the field name to the dataset name.

For example, the following code sets the *ReadOnly* property for the *CityStateZip* field in the *Customers* table to *True*:

```
CustomersCityStateZip.ReadOnly  := True;
```

And this statement changes field ordering by setting the *Index* property of the *CityStateZip* field in the *Customers* table to 3:

```
CustomersCityStateZip.Index := 3;
```

## Creating attribute sets for field components

When several fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), it is more convenient to set the properties for a single field, then store those properties as an attribute set in the Data Dictionary. Attribute sets stored in the data dictionary can be easily applied to other fields.

**Note** Attribute sets and the Data Dictionary are only available for BDE-enabled datasets.

To create an attribute set based on a field component in a dataset:

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to set properties.
3. Set the desired properties for the field in the Object Inspector.
4. Right-click the Fields editor list box to invoke the context menu.
5. Choose Save Attributes to save the current field's property settings as an attribute set in the Data Dictionary.

The name for the attribute set defaults to the name of the current field. You can specify a different name for the attribute set by choosing Save Attributes As instead of Save Attributes from the context menu.

Once you have created a new attribute set and added it to the Data Dictionary, you can then associate it with other persistent field components. Even if you later remove the association, the attribute set remains defined in the Data Dictionary.

**Note** You can also create attribute sets directly from the SQL Explorer. When you create an attribute set using SQL Explorer, it is added to the Data Dictionary, but not applied to any fields. SQL Explorer lets you specify two additional attributes: a field type (such as *TFloatField*, *TStringField*, and so on) and a data-aware control (such as *TDBEdit*, *TDBCheckBox*, and so on) that are automatically placed on a form when a field based on the attribute set is dragged to the form. For more information, see the online help for the SQL Explorer.

### Associating attribute sets with field components

When several fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), and you have saved those property settings as attribute sets in the Data Dictionary, you can easily apply the attribute sets to fields without having to recreate the settings manually for each field. In addition, if you later change the attribute settings in the Data Dictionary, those changes are automatically applied to every field associated with the set the next time field components are added to the dataset.

To apply an attribute set to a field component:

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to apply an attribute set.
3. Invoke the context menu and choose Associate Attributes.
4. Select or enter the attribute set to apply from the Associate Attributes dialog box. If there is an attribute set in the Data Dictionary that has the same name as the current field, that set name appears in the edit box.

**Important** If the attribute set in the Data Dictionary is changed at a later date, you must reapply the attribute set to each field component that uses it. You can invoke the Fields editor and multi-select field components within a dataset when reapplying attributes.

### Removing attribute associations

If you change your mind about associating an attribute set with a field, you can remove the association by following these steps:

1. Invoke the Fields editor for the dataset containing the field.
2. Select the field or fields from which to remove the attribute association.
3. Invoke the context menu for the Fields editor and choose Unassociate Attributes.

**Important** Unassociating an attribute set does not change any field properties. A field retains the settings it had when the attribute set was applied to it. To change these properties, select the field in the Fields editor and set its properties in the Object Inspector.

### Controlling and masking user input

The *EditMask* property provides a way to control the type and range of values a user can enter into a data-aware component associated with *TStringField*, *TDateField*, *TTimeField*, and *TDateTimeField*, and *TSQLTimeStampField* components. You can use existing masks or create your own. The easiest way to use and create edit masks is with the Input Mask editor. You can, however, enter masks directly into the *EditMask* field in the Object Inspector.

**Note** For *TStringField* components, the *EditMask* property is also its display format.

To invoke the Input Mask editor for a field component:

1. Select the component in the Fields editor or Object Inspector.
2. Click the Properties page in the Object Inspector.
3. Double-click the values column for the EditMask field in the Object Inspector, or click the ellipsis button. The Input Mask editor opens.

The Input Mask edit box lets you create and edit a mask format. The Sample Masks grid lets you select from predefined masks. If you select a sample mask, the mask format appears in the Input Mask edit box where you can modify it or use it as is. You can test the allowable user input for a mask in the Test Input edit box.

The Masks button enables you to load a custom set of masks—if you have created one—into the Sample Masks grid for easy selection.

### Using default formatting for numeric, date, and time fields

Delphi provides built-in display and edit format routines and intelligent default formatting for *TFloatField*, *TCurrencyField*, *TBCDField*, *TFMTBCDField*, *TIntegerField*, *TSmallIntField*, *TWordField*, *TDateField*, *TDateTimeField*, and *TTimeField*, and *TSQLTimeStampField* components. To use these routines, you need do nothing.

Default formatting is performed by the following routines:

**Table 25.4** Field component formatting routines

| Routine | Used by . . . |
|---|---|
| *FormatFloat* | *TFloatField, TCurrencyField* |
| *FormatDateTime* | *TDateField, TTimeField, TDateTimeField,* |
| SQLTimeStampToString | *TSQLTimeStampField* |
| *FormatCurr* | *TCurrencyField, TBCDField* |
| BcdToStrF | TFMTBCDField |

Only format properties appropriate to the data type of a field component are available for a given component.

Default formatting conventions for date, time, currency, and numeric values are based on the Regional Settings properties in the Control Panel. For example, using the default settings for the United States, a *TFloatField* column with the *Currency* property set to *True* sets the *DisplayFormat* property for the value 1234.56 to $1234.56, while the *EditFormat* is 1234.56.

At design time or runtime, you can edit the *DisplayFormat* and *EditFormat* properties of a field component to override the default display settings for that field. You can also write *OnGetText* and *OnSetText* event handlers to do custom formatting for field components at runtime.

## Handling events

Like most components, field components have events associated with them. Methods can be assigned as handlers for these events. By writing these handlers you can react to the occurrence of events that affect data entered in fields through data-aware controls and perform actions of your own design. The following table lists the events associated with field components:

**Table 25.5** Field component events

| Event | Purpose |
|---|---|
| *OnChange* | Called when the value for a field changes. |
| *OnGetText* | Called when the value for a field component is retrieved for display or editing. |
| *OnSetText* | Called when the value for a field component is set. |
| *OnValidate* | Called to validate the value for a field component whenever the value is changed because of an edit or insert operation. |

*OnGetText* and *OnSetText* events are primarily useful to programmers who want to do custom formatting that goes beyond the built-in formatting functions. *OnChange* is useful for performing application-specific tasks associated with data change, such as enabling or disabling menus or visual controls. *OnValidate* is useful when you want to control data-entry validation in your application before returning values to a database server.

To write an event handler for a field component:

1. Select the component.
2. Select the Events page in the Object Inspector.
3. Double-click the Value field for the event handler to display its source code window.
4. Create or edit the handler code.

## Working with field component methods at runtime

Field components methods available at runtime enable you to convert field values from one data type to another, and enable you to set focus to the first data-aware control in a form that is associated with a field component.

Controlling the focus of data-aware components associated with a field is important when your application performs record-oriented data validation in a dataset event handler (such as *BeforePost*). Validation may be performed on the fields in a record whether or not its associated data-aware control has focus. Should validation fail for a particular field in the record, you want the data-aware control containing the faulty data to have focus so that the user can enter corrections.

You control focus for a field's data-aware components with a field's *FocusControl* method. *FocusControl* sets focus to the first data-aware control in a form that is associated with a field. An event handler should call a field's *FocusControl* method before validating the field. The following code illustrates how to call the *FocusControl* method for the *Company* field in the *Customers* table:

```
CustomersCompany.FocusControl;
```

The following table lists some other field component methods and their uses. For a complete list and detailed information about using each method, see the entries for *TField* and its descendants in the online *VCL Reference*.

**Table 25.6** Selected field component methods

| Method | Purpose |
|---|---|
| AssignValue | Sets a field value to a specified value using an automatic conversion function based on the field's type. |
| Clear | Clears the field and sets its value to NULL. |
| GetData | Retrieves unformatted data from the field. |
| IsValidChar | Determines if a character entered by a user in a data-aware control to set a value is allowed for this field. |
| SetData | Assigns unformatted data to this field. |

## Displaying, converting, and accessing field values

Data-aware controls such as *TDBEdit* and *TDBGrid* automatically display the values associated with field components. If editing is enabled for the dataset and the controls, data-aware controls can also send new and changed values to the database. In general, the built-in properties and methods of data-aware controls enable them to connect to datasets, display values, and make updates without requiring extra programming on your part. Use them whenever possible in your database applications. For more information about data-aware control, see Chapter 20, "Using data controls."

Standard controls can also display and edit database values associated with field components. Using standard controls, however, may require additional programming on your part. For example, when using standard controls, your application is responsible for tracking when to update controls because field values change. If the dataset has a datasource component, you can use its events to help you do this. In particular, the *OnDataChange* event lets you know when you may need to update a control's value and the *OnStateChange* event can help you determine when to enable or disable controls. For more information on these events, see "Responding to changes mediated by the data source" on page 20-4.

The following topics discuss how to work with field values so that you can display them in standard controls.

### Displaying field component values in standard controls

An application can access the value of a dataset column through the *Value* property of a field component. For example, the following *OnDataChange* event handler updates the text in a *TEdit* control because the value of the *CustomersCompany* field may have changed:

```
procedure TForm1.CustomersDataChange(Sender: TObject, Field: TField);
begin
  Edit3.Text := CustomersCompany.Value;
end;
```

This method works well for string values, but may require additional programming to handle conversions for other data types. Fortunately, field components have built-in properties for handling conversions.

**Note** You can also use Variants to access and set field values. For more information about using variants to access and set field values, see "Accessing field values with the default dataset property" on page 25-20.

## Converting field values

Conversion properties attempt to convert one data type to another. For example, the *AsString* property converts numeric and Boolean values to string representations. The following table lists field component conversion properties, and which properties are recommended for field components by field-component class:

| | AsVariant | AsString | AsInteger | AsFloat AsCurrency AsBCD | AsDateTime AsSQLTimeStamp | AsBoolean |
|---|---|---|---|---|---|---|
| TStringField | yes | NA | yes | yes | yes | yes |
| TWideStringField | yes | yes | yes | yes | yes | yes |
| TIntegerField | yes | yes | NA | yes | | |
| TSmallIntField | yes | yes | yes | yes | | |
| TWordField | yes | yes | yes | yes | | |
| TLargeintField | yes | yes | yes | yes | | |
| TFloatField | yes | yes | yes | yes | | |
| TCurrencyField | yes | yes | yes | yes | | |
| TBCDField | yes | yes | yes | yes | | |
| TFMTBCDField | yes | yes | yes | yes | | |
| TDateTimeField | yes | yes | | yes | yes | |
| TDateField | yes | yes | | yes | yes | |
| TTimeField | yes | yes | | yes | yes | |
| TSQLTimeStampField | yes | yes | | yes | yes | |
| TBooleanField | yes | yes | | | | |
| TBytesField | yes | yes | | | | |
| TVarBytesField | yes | yes | | | | |
| TBlobField | yes | yes | | | | |
| TMemoField | yes | yes | | | | |
| TGraphicField | yes | yes | | | | |
| TVariantField | NA | yes | yes | yes | yes | yes |
| TAggregateField | yes | yes | | | | |

Note that some columns in the table refer to more than one conversion property (such as *AsFloat*, *AsCurrency*, and *AsBCD*). This is because all field data types that support one of those properties always support the others as well.

Note also that the *AsVariant* property can translate among all data types. For any datatypes not listed above, *AsVariant* is also available (and is, in fact, the only option). When in doubt, use *AsVariant*.

In some cases, conversions are not always possible. For example, *AsDateTime* can be used to convert a string to a date, time, or datetime format only if the string value is in a recognizable datetime format. A failed conversion attempt raises an exception.

In some other cases, conversion is possible, but the results of the conversion are not always intuitive. For example, what does it mean to convert a *TDateTimeField* value into a float format? *AsFloat* converts the date portion of the field to the number of days since 12/31/1899, and it converts the time portion of the field to a fraction of 24 hours. Table 25.7 lists permissible conversions that produce special results:

**Table 25.7** Special conversion results

| Conversion | Result |
|---|---|
| *String to Boolean* | Converts "True," "False," "Yes," and "No" to Boolean. Other values raise exceptions. |
| *Float to Integer* | Rounds float value to nearest integer value. |
| *DateTime or SQLTimeStamp to Float* | Converts date to number of days since 12/31/1899, time to a fraction of 24 hours. |
| *Boolean to String* | Converts any Boolean value to "True" or "False." |

In other cases, conversions are not possible at all. In these cases, attempting a conversion also raises an exception.

Conversion always occurs before an assignment is made. For example, the following statement converts the value of *CustomersCustNo* to a string and assigns the string to the text of an edit control:

```
Edit1.Text := CustomersCustNo.AsString;
```

Conversely, the next statement assigns the text of an edit control to the *CustomersCustNo* field as an integer:

```
MyTableMyField.AsInteger := StrToInt(Edit1.Text);
```

## Accessing field values with the default dataset property

The most general method for accessing a field's value is to use Variants with the *FieldValues* property. For example, the following statement puts the value of an edit box into the *CustNo* field in the *Customers* table:

```
Customers.FieldValues['CustNo'] := Edit2.Text;
```

Because the *FieldValues* property is of type Variant, it automatically converts other datatypes into a Variant value.

For more information about Variants, see the online help.

## Accessing field values with a dataset's Fields property

You can access the value of a field with the *Fields* property of the dataset component to which the field belongs. *Fields* maintains an indexed list of all the fields in the dataset. Accessing field values with the *Fields* property is useful when you need to iterate over a number of columns, or if your application works with tables that are not available to you at design time.

To use the *Fields* property you must know the order of and data types of fields in the dataset. You use an ordinal number to specify the field to access. The first field in a dataset is numbered 0. Field values must be converted as appropriate using each field component's conversion properties. For more information about field component conversion properties, see "Converting field values" on page 25-19.

For example, the following statement assigns the current value of the seventh column (Country) in the *Customers* table to an edit control:

```
Edit1.Text := CustTable.Fields[6].AsString;
```

Conversely, you can assign a value to a field by setting the *Fields* property of the dataset to the desired field. For example:

```
begin
  Customers.Edit;
  Customers.Fields[6].AsString := Edit1.Text;
  Customers.Post;
end;
```

## Accessing field values with a dataset's FieldByName method

You can also access the value of a field with a dataset's *FieldByName* method. This method is useful when you know the name of the field you want to access, but do not have access to the underlying table at design time.

To use *FieldByName*, you must know the dataset and name of the field you want to access. You pass the field's name as an argument to the method. To access or change the field's value, convert the result with the appropriate field component conversion property, such as *AsString* or *AsInteger*. For example, the following statement assigns the value of the *CustNo* field in the *Customers* dataset to an edit control:

```
Edit2.Text := Customers.FieldByName('CustNo').AsString;
```

Conversely, you can assign a value to a field:

```
begin
  Customers.Edit;
  Customers.FieldByName('CustNo').AsString := Edit2.Text;
  Customers.Post;
end;
```

## Setting a default value for a field

You can specify how a default value for a field in a client dataset or a BDE-enabled dataset should be calculated at runtime using the *DefaultExpression* property. *DefaultExpression* can be any valid SQL value expression that does not refer to field values. If the expression contains literals other than numeric values, they must appear in quotes. For example, a default value of noon for a time field would be

```
'12:00:00'
```

including the quotes around the literal value.

**Note** If the underlying database table defines a default value for the field, the default you specify in *DefaultExpression* takes precedence. That is because *DefaultExpression* is applied when the dataset posts the record containing the field, before the edited record is applied to the database server.

## Working with constraints

Field components in client datasets or BDE-enabled datasets can use SQL server constraints. In addition, your applications can create and use custom constraints for these datasets that are local to your application. All constraints are rules or conditions that impose a limit on the scope or range of values that a field can store.

### Creating a custom constraint

A custom constraint is not imported from the server like other constraints. It is a constraint that you declare, implement, and enforce in your local application. As such, custom constraints can be useful for offering a prevalidation enforcement of data entry, but a custom constraint cannot be applied against data received from or sent to a server application.

To create a custom constraint, set the *CustomConstraint* property to specify a constraint condition, and set *ConstraintErrorMessage* to the message to display when a user violates the constraint at runtime.

*CustomConstraint* is an SQL string that specifies any application-specific constraints imposed on the field's value. Set *CustomConstraint* to limit the values that the user can enter into a field. *CustomConstraint* can be any valid SQL search expression such as

```
x > 0 and x < 100
```

The name used to refer to the value of the field can be any string that is not a reserved SQL keyword, as long as it is used consistently throughout the constraint expression.

**Note** Custom constraints are only available in BDE-enabled and client datasets.

Custom constraints are imposed in addition to any constraints to the field's value that come from the server. To see the constraints imposed by the server, read the *ImportedConstraint* property.

## Using server constraints

Most production SQL databases use constraints to impose conditions on the possible values for a field. For example, a field may not permit NULL values, may require that its value be unique for that column, or that its values be greater than *0* and less than *150*. While you could replicate such conditions in your client applications, client datasets and BDE-enabled datasets offer the *ImportedConstraint* property to propagate a server's constraints locally.

*ImportedConstraint* is a read-only property that specifies an SQL clause that limits field values in some manner. For example:

```
Value > 0 and Value < 100
```

Do not change the value of *ImportedConstraint*, except to edit nonstandard or server-specific SQL that has been imported as a comment because it cannot be interpreted by the database engine.

To add additional constraints on the field value, use the *CustomConstraint* property. Custom constraints are imposed in addition to the imported constraints. If the server constraints change, the value of *ImportedConstraint* also changed but constraints introduced in the *CustomConstraint* property persist.

Removing constraints from the *ImportedConstraint* property will not change the validity of field values that violate those constraints. Removing constraints results in the constraints being checked by the server instead of locally. When constraints are checked locally, the error message supplied as the *ConstraintErrorMessage* property is displayed when violations are found, instead of displaying an error message from the server.

## Using object fields

Object fields are fields that represent a composite of other, simpler datatypes. These include ADT (Abstract Data Type) fields, Array fields, DataSet fields, and Reference fields. All of these field types either contain or reference child fields or other data sets.

ADT fields and array fields are fields that contain child fields. The child fields of an ADT field can be any scalar or object type (that is, any other field type). These child fields may differ in type from each other. An array field contains an array of child fields, all of the same type.

Dataset and reference fields are fields that access other data sets. A dataset field provides access to a nested (detail) dataset and a reference field stores a pointer (reference) to another persistent object (ADT).

**Table 25.8** Types of object field components

| Component name | Purpose |
|---|---|
| TADTField | Represents an ADT (Abstract Data Type) field. |
| TArrayField | Represents an array field. |
| TDataSetField | Represents a field that contains a nested data set reference. |
| TReferenceField | Represents a REF field, a pointer to an ADT. |

When you add fields with the Fields editor to a dataset that contains object fields, persistent object fields of the correct type are automatically created for you. Adding persistent object fields to a dataset automatically sets the dataset's *ObjectView* property to *True*, which instructs the dataset to store these fields hierarchically, rather than flattening them out as if the constituent child fields were separate, independent fields.

The following properties are common to all object fields and provide the functionality to handle child fields and datasets.

**Table 25.9** Common object field descendant properties

| Property | Purpose |
|---|---|
| Fields | Contains the child fields belonging to the object field. |
| ObjectType | Classifies the object field. |
| FieldCount | Number of child fields belonging to the object field. |
| FieldValues | Provides access to the values of the child fields. |

## Displaying ADT and array fields

Both ADT and array fields contain child fields that can be displayed through data-aware controls.

Data-aware controls such as *TDBEdit* that represent a single field value display child field values in an uneditable comma delimited string. In addition, if you set the control's *DataField* property to the child field instead of the object field itself, the child field can be viewed an edited just like any other normal data field.

A *TDBGrid* control displays ADT and array field data differently, depending on the value of the dataset's *ObjectView* property. When *ObjectView* is *False*, each child field appears in a single column. When *ObjectView* is *True*, an ADT or array field can be expanded and collapsed by clicking on the arrow in the title bar of the column. When the field is expanded, each child field appears in its own column and title bar, all below the title bar of the ADT or array itself. When the ADT or array is collapsed, only one column appears with an uneditable comma-delimited string containing the child fields.

## Working with ADT fields

ADTs are user-defined types created on the server, and are similar to the record type. An ADT can contain most scalar field types, array fields, reference fields, and nested ADTs.

There are a variety of ways to access the data in ADT field types. These are illustrated in the following examples, which assign a child field value to an edit box called *CityEdit*, and use the following ADT structure,

```
Address
  Street
  City
  State
  Zip
```

### Using persistent field components

The easiest way to access ADT field values is to use persistent field components. For the ADT structure above, the following persistent fields can be added to the *Customer* table using the Fields editor:

```
CustomerAddress: TADTField;
CustomerAddrStreet: TStringField;
CustomerAddrCity: TStringField;
CustomerAddrState: TStringField;
CustomerAddrZip: TStringField;
```

Given these persistent fields, you can simply access the child fields of an ADT field by name:

```
CityEdit.Text := CustomerAddrCity.AsString;
```

Although persistent fields are the easiest way to access ADT child fields, it is not possible to use them if the structure of the dataset is not known at design time. When accessing ADT child fields without using persistent fields, you must set the dataset's *ObjectView* property to *True*.

### Using the dataset's FieldByName method

You can access the children of an ADT field using the dataset's *FieldByName* method by qualifying the name of the child field with the ADT field's name:

```
CityEdit.Text := Customer.FieldByName('Address.City').AsString;
```

### Using the dateset's FieldValues property

You can also use qualified field names with a dataset's *FieldValues* property:

```
CityEdit.Text := Customer['Address.City'];
```

Note that you can omit the property name (*FieldValues*) because *FieldValues* is the dataset's default property.

**Note** Unlike other runtime methods for accessing ADT child field values, the *FieldValues* property works even if the dataset's *ObjectView* property is *False*.

### Using the ADT field's FieldValues property

You can access the value of a child field with the *TADTField*'s *FieldValues* property. *FieldValues* accepts and returns a *Variant*, so it can handle and convert fields of any type. The index parameter is an integer value that specifies the offset of the field.

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).FieldValues[1];
```

Because *FieldValues* is the default property of *TADTField*, the property name (*FieldValues*) can be omitted. Thus, the following statement is equivalent to the one above:

```
CityEdit.Text := TADTField(Customer.FieldByName('Address'))[1];
```

### Using the ADT field's Fields property

Each ADT field has a *Fields* property that is analogous to the *Fields* property of a dataset. Like the *Fields* property of a dataset, you can use it to access child fields by position:

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).Fields[1].AsString;
```

or by name:

```
CityEdit.Text :=
TADTField(Customer.FieldByName('Address')).Fields.FieldByName('City').AsString;
```

## Working with array fields

Array fields consist of a set of fields of the same type. The field types can be scalar (for example, float, string), or non-scalar (an ADT), but an array field of arrays is not permitted. The SparseArrays property of *TDataSet* determines whether a unique *TField* object is created for each element of the array field.

There are a variety of ways to access the data in array field types. If you are not using persistent fields, the dataset's *ObjectView* property must be set to *True* before you can access the elements of an array field.

### Using persistent fields

You can map persistent fields to the individual array elements in an array field. For example, consider an array field *TelNos_Array*, which is a six element array of strings. The following persistent fields created for the *Customer* table component represent the *TelNos_Array* field and its six elements:

```
CustomerTelNos_Array: TArrayField;
CustomerTelNos_Array0: TStringField;
CustomerTelNos_Array1: TStringField;
CustomerTelNos_Array2: TStringField;
CustomerTelNos_Array3: TStringField;
CustomerTelNos_Array4: TStringField;
CustomerTelNos_Array5: TStringField;
```

Given these persistent fields, the following code uses a persistent field to assign an array element value to an edit box named *TelEdit*.

```
TelEdit.Text := CustomerTelNos_Array0.AsString;
```

### Using the array field's FieldValues property

You can access the value of a child field with the array field's *FieldValues* property. *FieldValues* accepts and returns a *Variant*, so it can handle and convert child fields of any type. For example,

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array')).FieldValues[1];
```

Because *FieldValues* is the default property of *TArrayField*, this can also be written

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array'))[1];
```

### Using the array field's Fields property

*TArrayField* has a *Fields* property that you can use to access individual sub-fields. This is illustrated below, where an array field (*OrderDates*) is used to populate a list box with all non-null array elements:

```
for I := 0 to OrderDates.Size - 1 do
begin
  if not OrderDates.Fields[I].IsNull then
    OrderDateListBox.Items.Add(OrderDates[I]);
end;
```

## Working with dataset fields

Dataset fields provide access to data stored in a nested dataset. The NestedDataSet property references the nested dataset. The data in the nested dataset is then accessed through the field objects of the nested dataset.

### Displaying dataset fields

*TDBGrid* controls enable the display of data stored in data set fields. In a *TDBGrid* control, a dataset field is indicated in each cell of a dataset column with the string "(DataSet)", and at runtime an ellipsis button also exists to the right. Clicking on the ellipsis brings up a new form with a grid displaying the dataset associated with the current record's dataset field. This form can also be brought up programmatically with the DB grid's *ShowPopupEditor* method. For example, if the seventh column in the grid represents a dataset field, the following code will display the dataset associated with that field for the current record.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

### Accessing data in a nested dataset

A dataset field is not normally bound directly to a data aware control. Rather, since a nested data set is just that, a data set, the means to get at its data is via a *TDataSet* descendant. The type of dataset you use is determined by the parent dataset (the one with the dataset field.) For example, a BDE-enabled dataset uses *TNestedTable* to represent the data in its dataset fields, while client datasets use other client datasets.

To access the data in a dataset field,

1 Create a persistent *TDataSetField* object by invoking the Fields editor for the parent dataset.

2 Create a dataset to represent the values in that dataset field. It must be of a type compatible with the parent dataset.

3 Set that *DataSetField* property of the dataset created in step 2 to the persistent dataset field you created in step 1.

If the nested dataset field for the current record has a value, the detail dataset component will contain records with the nested data; otherwise, the detail dataset will be empty.

Before inserting records into a nested dataset, you should be sure to post the corresponding record in the master table, if it has just been inserted. If the inserted record is not posted, it will be automatically posted before the nested dataset posts.

## Working with reference fields

Reference fields store a pointer or reference to another ADT object. This ADT object is a single record of another object table. Reference fields always refer to a single record in a dataset (object table). The data in the referenced object is actually returned in a nested dataset, but can also be accessed via the *Fields* property on the *TReferenceField*.

### Displaying reference fields

In a *TDBGrid* control a reference field is designated in each cell of the dataset column, with (Reference) and, at runtime, an ellipsis button to the right. At runtime, clicking on the ellipsis brings up a new form with a grid displaying the object associated with the current record's reference field.

This form can also be brought up programmatically with the DB grid's *ShowPopupEditor* method. For example, if the seventh column in the grid represents a reference field, the following code will display the object associated with that field for the current record.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

## Accessing data in a reference field

You can access the data in a reference field in the same way you access a nested dataset:

**1** Create a persistent *TDataSetField* object by invoking the Fields editor for the parent dataset.

**2** Create a dataset to represent the value of that dataset field.

**3** Set that *DataSetField* property of the dataset created in step 2 to the persistent dataset field you created in step 1.

If the reference is assigned, the reference dataset will contain a single record with the referenced data. If the reference is null, the reference dataset will be empty.

You can also use the reference field's Fields property to access the data in a reference field. For example, the following lines are equivalent and assign data from the reference field *CustomerRefCity* to an edit box called *CityEdit*:

```
CityEdit.Text  := CustomerRefCity.Fields[1].AsString;
CityEdit.Text  := CustomerRefCity.NestedDataSet.Fields[1].AsString;
```

When data in a reference field is edited, it is actually the referenced data that is modified.

To assign a reference field, you need to first use a SELECT statement to select the reference from the table, and then assign. For example:

```
var
  AddressQuery: TQuery;
  CustomerAddressRef: TReferenceField;
begin
  AddressQuery.SQL.Text := 'SELECT REF(A) FROM AddressTable A WHERE A.City = ''San
  Francisco''';
  AddressQuery.Open;
  CustomerAddressRef.Assign(AddressQuery.Fields[0]);
end;
```

Chapter 26

# Using the Borland Database Engine

The Borland Database Engine (BDE) is a data-access mechanism that can be shared by several applications. The BDE defines a powerful library of API calls that can create, restructure, fetch data from, update, and otherwise manipulate local and remote database servers. The BDE provides a uniform interface to access a wide variety of database servers, using drivers to connect to different databases. Depending on your version of Delphi, you can use the drivers for local databases (Paradox, dBASE, FoxPro, and Access), SQL Links drivers for remote database servers such as InterBase, Oracle, Sybase, Informix, Microsoft SQL server, and DB2, and an ODBC adapter that lets you supply your own ODBC drivers.

When deploying BDE-based applications, you must include the BDE with your application. While this increases the size of the application and the complexity of deployment, the BDE can be shared with other BDE-based applications and provides a broad range of support for database manipulation. Although you can use the BDE's API directly in your application, the components on the BDE page of the Component palette wrap most of this functionality for you.

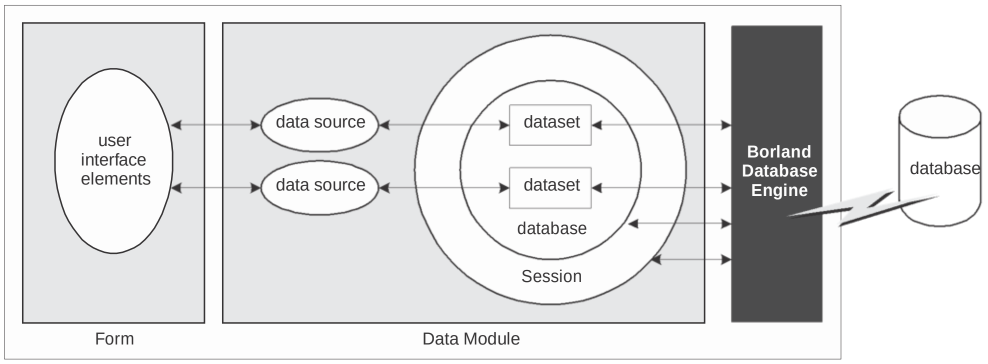
## BDE-based architecture

When using the BDE, your application uses a variation of the general database architecture described in "Database architecture" on page 19-6. In addition to the user interface elements, datasource, and datasets common to all Delphi database applications, A BDE-based application can include

- One or more database components to control transactions and to manage database connections.
- One or more session components to isolate data access operations such as database connections, and to manage groups of databases.

The relationships between the components in a BDE-based application are illustrated in Figure 26.1:

**Figure 26.1** Components in a BDE-based application



## Using BDE-enabled datasets

BDE-enabled datasets use the Borland Database Engine (BDE) to access data. They inherit the common dataset capabilities described in Chapter 24, "Understanding datasets," using the BDE to provide the implementation. In addition, all BDE datasets add properties, events, and methods for

- Associating a dataset with database and session connections.
- Caching BLOBs.
- Obtaining a BDE handle.

There are three BDE-enabled datasets:

- *TTable*, a table type dataset that represents all of the rows and columns of a single database table. See "Using table type datasets" on page 24-25 for a description of features common to table type datasets. See "Using TTable" on page 26-5 for a description of features unique to *TTable*.
- *TQuery*, a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any. See "Using query-type datasets" on page 24-42 for a description of features common to query-type datasets. See "Using TQuery" on page 26-9 for a description of features unique to *TQuery*.
- *TStoredProc*, a stored procedure-type dataset that executes a stored procedure that is defined on a database server. See "Using stored procedure-type datasets" on page 24-50 for a description of features common to stored procedure-type datasets. See "Using TStoredProc" on page 26-11 for a description of features unique to *TStoredProc*.

**Note** In addition to the three types of BDE-enabled datasets, there is a BDE-based client dataset (*TBDEClientDataSet*) that can be used for caching updates. For information on caching updates, see "Using a client dataset to cache updates" on page 29-16.

### Associating a dataset with database and session connections

In order for a BDE-enabled dataset to fetch data from a database server it needs to use both a database and a session.

- Databases represent connections to specific database servers. The database identifies a BDE driver, a particular database server that uses that driver, and a set of connection parameters for connecting to that database server. Each database is represented by a *TDatabase* component. You can either associate your datasets with a *TDatabase* component you add to a form or data module, or you can simply identify the database server by name and let Delphi generate an implicit database component for you. Using an explicitly-created *TDatabase* component is recommended for most applications, because the database component gives you greater control over how the connection is established, including the login process, and lets you create and use transactions.

  To associate a BDE-enabled dataset with a database, use the *DatabaseName* property. DatabaseName is a string that contains different information, depending on whether you are using an explicit database component and, if not, the type of database you are using:

  - If you are using an explicit *TDatabase* component, *DatabaseName* is the value of the *DatabaseName* property of the database component.
  - If you are want to use an implicit database component and the database has a BDE alias, you can specify a BDE alias as the value of *DatabaseName*. A BDE alias represents a database plus configuration information for that database. The configuration information associated with an alias differs by database type (Oracle, Sybase, InterBase, Paradox, dBASE, and so on). Use the BDE Administration tool or the SQL explorer to create and manage BDE aliases.
  - If you want to use an implicit database component for a Paradox or dBASE database, you can also use *DatabaseName* to simply specify the directory where the database tables are located.

- A session provides global management for a group of database connections in an application. When you add BDE-enabled datasets to your application, your application automatically contains a session component, named *Session*. As you add database and dataset components to the application, they are automatically associated with this default session. It also controls access to password protected Paradox files, and it specifies directory locations for sharing Paradox files over a network. You can control database connections and access to Paradox files using the properties, events, and methods of the session.

  You can use the default session to control all database connections in your application. Alternatively, you can add additional session components at design time or create them dynamically at runtime to control a subset of database connections in an application. To associate your dataset with an explicitly created session component, use the *SessionName* property. If you do not use explicit session components in your application, you do not have to provide a value for this property. Whether you use the default session or explicitly specify a session using the *SessionName* property, you can access the session associated with a dataset by reading the *DBSession* property.

**Note** If you use a session component, the *SessionName* property of a dataset must match the *SessionName* property for the database component with which the dataset is associated.

For more information about *TDatabase* and *TSession*, see "Connecting to databases with TDatabase" on page 26-12 and "Managing database sessions" on page 26-16.

### Caching BLOBs

BDE-enabled datasets all have a *CacheBlobs* property that controls whether BLOB fields are cached locally by the BDE when an application reads BLOB records. By default, *CacheBlobs* is *True*, meaning that the BDE caches a local copy of BLOB fields. Caching BLOBs improves application performance by enabling the BDE to store local copies of BLOBs instead of fetching them repeatedly from the database server as a user scrolls through records.

In applications and environments where BLOBs are frequently updated or replaced, and a fresh view of BLOB data is more important than application performance, you can set *CacheBlobs* to *False* to ensure that your application always sees the latest version of a BLOB field.

### Obtaining a BDE handle

You can use BDE-enabled datasets without ever needing to make direct API calls to the Borland Database Engine. The BDE-enabled datasets, in combination with database and session components, encapsulate much of the BDE functionality. However, if you need to make direct API calls to the BDE, you may need BDE handles for resources managed by the BDE. Many BDE APIs require these handles as parameters.

All BDE-enabled datasets include three read-only properties for accessing BDE handles at runtime:

- *Handle* is a handle to the BDE cursor that accesses the records in the dataset.
- *DBHandle* is a handle to the database that contains the underlying tables or stored procedure.
- *DBLocale* is a handle to the BDE language driver for the dataset. The locale controls the sort order and character set used for string data.

These properties are automatically assigned to a dataset when it is connected to a database server through the BDE.

## Using TTable

*TTable* encapsulates the full structure of and data in an underlying database table. It implements all of the basic functionality introduced by *TDataSet*, as well as all of the special features typical of table type datasets. Before looking at the unique features introduced by *TTable*, you should familiarize yourself with the common database features described in "Understanding datasets," including the section on table type datasets that starts on page 24-25.

Because *TTable* is a BDE-enabled dataset, it must be associated with a database and a session. "Associating a dataset with database and session connections" on page 26-3 describes how you form these associations. Once the dataset is associated with a database and session, you can bind it to a particular database table by setting the *TableName* property and, if you are using a Paradox, dBASE, FoxPro, or comma-delimited ASCII text table, the *TableType* property.

**Note** The table must be closed when you change its association to a database, session, or database table, or when you set the *TableType* property. However, before you close the table to change these properties, first post or discard any pending changes. If cached updates are enabled, call the *ApplyUpdates* method to write the posted changes to the database.

*TTable* components are unique in the support they offer for local database tables (Paradox, dBASE, FoxPro, and comma-delimited ASCII text tables). The following topics describe the special properties and methods that implement this support.

In addition, *TTable* components can take advantage of the BDE's support for batch operations (table level operations to append, update, delete, or copy entire groups of records). This support is described in "Importing data from another table" on page 26-8.

### Specifying the table type for local tables

If an application accesses Paradox, dBASE, FoxPro, or comma-delimited ASCII text tables, then the BDE uses the *TableType* property to determine the table's type (its expected structure). *TableType* is not used when *TTable* represents an SQL-based table on a database server.

By default *TableType* is set to *ttDefault*. When *TableType* is *ttDefault*, the BDE determines a table's type from its filename extension. Table 26.1 summarizes the file extensions recognized by the BDE and the assumptions it makes about a table's type:

**Table 26.1** Table types recognized by the BDE based on file extension

| Extension | Table type |
|---|---|
| No file extension | Paradox |
| .DB | Paradox |
| .DBF | dBASE |
| .TXT | ASCII text |

If your local Paradox, dBASE, and ASCII text tables use the file extensions as described in Table 26.1, then you can leave *TableType* set to *ttDefault*. Otherwise, your application must set *TableType* to indicate the correct table type. Table 26.2 indicates the values you can assign to *TableType*:

**Table 26.2** TableType values

| Value | Table type |
|---|---|
| ttDefault | Table type determined automatically by the BDE |
| ttParadox | Paradox |
| ttDBase | dBASE |
| ttFoxPro | FoxPro |
| ttASCII | Comma-delimited ASCII text |

## Controlling read/write access to local tables

Like any table type dataset, *TTable* lets you control read and write access by your application using the *ReadOnly* property.

In addition, for Paradox, dBASE, and FoxPro tables, *TTable* can let you control read and write access to tables by other applications. The *Exclusive* property controls whether your application gains sole read/write access to a Paradox, dBASE, or FoxPro table. To gain sole read/write access for these table types, set the table component's *Exclusive* property to *True* before opening the table. If you succeed in opening a table for exclusive access, other applications cannot read data from or write data to the table. Your request for exclusive access is not honored if the table is already in use when you attempt to open it.

The following statements open a table for exclusive access:

```
CustomersTable.Exclusive := True; {Set request for exclusive lock}
CustomersTable.Active := True; {Now open the table}
```

**Note** You can attempt to set *Exclusive* on SQL tables, but some servers do not support exclusive table-level locking. Others may grant an exclusive lock, but permit other applications to read data from the table. For more information about exclusive locking of database tables on your server, see your server documentation.

## Specifying a dBASE index file

For most servers, you use the methods common to all table type datasets to specify an index. These methods are described in "Sorting records with indexes" on page 24-26.

For dBASE tables that use non-production index files or dBASE III PLUS-style indexes (*.NDX), however, you must use the *IndexFiles and IndexName* properties instead. Set the *IndexFiles* property to the name of the non-production index file or list the .NDX files. Then, specify one index in the *IndexName* property to have it actively sorting the dataset.

At design time, click the ellipsis button in the *IndexFiles* property value in the Object Inspector to invoke the Index Files editor. To add one non-production index file or .NDX file: click the Add button in the Index Files dialog and select the file from the Open dialog. Repeat this process once for each non-production index file or .NDX file. Click the OK button in the Index Files dialog after adding all desired indexes.

This same operation can be performed programmatically at runtime. To do this, access the *IndexFiles* property using properties and methods of string lists. When adding a new set of indexes, first call the *Clear* method of the table's *IndexFiles* property to remove any existing entries. Call the *Add* method to add each non-production index file or .NDX file:

```
with Table2.IndexFiles do begin
  Clear;
  Add('Bystate.ndx');
  Add('Byzip.ndx');
  Add('Fullname.ndx');
  Add('St_name.ndx');
end;
```

After adding any desired non-production or .NDX index files, the names of individual indexes in the index file are available, and can be assigned to the *IndexName* property. The index tags are also listed when using the *GetIndexNames* method and when inspecting index definitions through the *TIndexDef* objects in the *IndexDefs* property. Properly listed .NDX files are automatically updated as data is added, changed, or deleted in the table (regardless of whether a given index is used in the *IndexName* property).

In the example below, the *IndexFiles* for the *AnimalsTable* table component is set to the non-production index file ANIMALS.MDX, and then its *IndexName* property is set to the index tag called "NAME":

```
AnimalsTable.IndexFiles.Add('ANIMALS.MDX');
AnimalsTable.IndexName := 'NAME';
```

Once you have specified the index file, using non-production or .NDX indexes works the same as any other index. Specifying an index name sorts the data in the table and makes it available for indexed-based searches, ranges, and (for non-production indexes) master-detail linking. See "Using table type datasets" on page 24-25 for details on these uses of indexes.

There are two special considerations when using dBASE III PLUS-style .NDX indexes with *TTable* components. The first is that .NDX files cannot be used as the basis for master-detail links. The second is that when activating a .NDX index with the *IndexName* property, you must include the .NDX extension in the property value as part of the index name:

```
with Table1 do begin
  IndexName := 'ByState.NDX';
  FindKey(['CA']);
end;
```

### Renaming local tables

To rename a Paradox or dBASE table at design time, right-click the table component and select Rename Table from the context menu.

To rename a Paradox or dBASE table at runtime, call the table's *RenameTable* method. For example, the following statement renames the Customer table to CustInfo:

```
Customer.RenameTable('CustInfo');
```

### Importing data from another table

You can use a table component's *BatchMove* method to import data from another table. *BatchMove* can

- Copy records from another table into this table.
- Update records in this table that occur in another table.
- Append records from another table to the end of this table.
- Delete records in this table that occur in another table.

*BatchMove* takes two parameters: the name of the table from which to import data, and a mode specification that determines which import operation to perform. Table 26.3 describes the possible settings for the mode specification:

**Table 26.3** BatchMove import modes

| Value | Meaning |
|---|---|
| batAppend | Append all records from the source table to the end of this table. |
| batAppendUpdate | Append all records from the source table to the end of this table and update existing records in this table with matching records from the source table. |
| batCopy | Copy all records from the source table into this table. |
| batDelete | Delete all records in this table that also appear in the source table. |
| batUpdate | Update existing records in this table with matching records from the source table. |

For example, the following code updates all records in the current table with records from the *Customer* table that have the same values for fields in the current index:

```
Table1.BatchMove('CUSTOMER.DB', batUpdate);
```

*BatchMove* returns the number of records it imports successfully.

**Caution** Importing records using the *batCopy* mode overwrites existing records. To preserve existing records use *batAppend* instead.

*BatchMove* performs only some of the batch operations supported by the BDE. Additional functions are available using the *TBatchMove* component. If you need to move a large amount of data between or among tables, use *TBatchMove* instead of calling a table's *BatchMove* method. For information about using *TBatchMove*, see "Using TBatchMove" on page 26-49.

## Using TQuery

*TQuery* represents a single Data Definition Language (DDL) or Data Manipulation Language (DML) statement (For example, a SELECT, INSERT, DELETE, UPDATE, CREATE INDEX, or ALTER TABLE command). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language. *TQuery* implements all of the basic functionality introduced by *TDataSet*, as well as all of the special features typical of query-type datasets. Before looking at the unique features introduced by *TQuery*, you should familiarize yourself with the common database features described in "Understanding datasets," including the section on query-type datasets that starts on page 24-42.

Because *TQuery* is a BDE-enabled dataset, it must usually be associated with a database and a session. (The one exception is when you use the *TQuery* for a heterogeneous query.) "Associating a dataset with database and session connections" on page 26-3 describes how you form these associations. You specify the SQL statement for the query by setting the *SQL* property.

*A TQuery* component can access data in:

- Paradox or dBASE tables, using Local SQL, which is part of the BDE. Local SQL is a subset of the SQL-92 specification. Most DML is supported and enough DDL syntax to work with these types of tables. See the local SQL help, LOCALSQL.HLP, for details on supported SQL syntax.
- Local InterBase Server databases, using the InterBase engine. For information on InterBase's SQL-92 standard SQL syntax support and extended syntax support, see the InterBase *Language Reference.*
- Databases on remote database servers such as Oracle, Sybase, MS-SQL Server, Informix, DB2, and InterBase. You must install the appropriate SQL Link driver and client software (vendor-supplied) specific to the database server to access a remote server. Any standard SQL syntax supported by these servers is allowed. For information on SQL syntax, limitations, and extensions, see the documentation for your particular server.

### Creating heterogeneous queries

*TQuery* supports heterogeneous queries against more than one server or table type (for example, data from an Oracle table and a Paradox table). When you execute a heterogeneous query, the BDE parses and processes the query using Local SQL. Because BDE uses Local SQL, extended, server-specific SQL syntax is not supported.

To perform a heterogeneous query, follow these steps:

1. Define separate BDE aliases for each database accessed in the query using the BDE BDE Administration tool or the SQL explorer.
2. Leave the *DatabaseName* property of the *TQuery* blank; the names of the databases used will be specified in the SQL statement.
3. In the SQL property, specify the SQL statement to execute. Precede each table name in the statement with the BDE alias for the table's database, enclosed in colons. This whole reference is then enclosed in quotation marks.

4 Set any parameters for the query in the *Params* property.

5 Call *Prepare* to prepare the query for execution prior to executing it for the first time.

6 Call *Open* or *ExecSQL* depending on the type of query you are executing.

For example, suppose you define an alias called *Oracle1* for an Oracle database that has a CUSTOMER table, and *Sybase1* for a Sybase database that has an ORDERS table. A simple query against these two tables would be:

```
SELECT Customer.CustNo, Orders.OrderNo
FROM ":Oracle1:CUSTOMER"
  JOIN ":Sybase1:ORDERS"
    ON (Customer.CustNo = Orders.CustNo)
WHERE (Customer.CustNo = 1503)
```

As an alternative to using a BDE alias to specify the database in a heterogeneous query, you can use a *TDatabase* component. Configure the *TDatabase* as normal to point to the database, set the *TDatabase.DatabaseName* to an arbitrary but unique value, and then use that value in the SQL statement instead of a BDE alias name.

## Obtaining an editable result set

To request a result set that users can edit in data-aware controls, set a query component's *RequestLive* property to *True*. Setting *RequestLive* to *True* does not guarantee a live result set, but the BDE attempts to honor the request whenever possible. There are some restrictions on live result set requests, depending on whether the query uses the local SQL parser or a server's SQL parser.

- Queries where table names are preceded by a BDE database alias (as in heterogeneous queries) and queries executed against Paradox or dBASE are parsed by the BDE using Local SQL. When queries use the local SQL parser, the BDE offers expanded support for updatable, live result sets in both single table and multi-table queries. When using Local SQL, a live result set for a query against a single table or view is returned if the query does not contain any of the following:
  - DISTINCT in the SELECT clause
  - Joins (inner, outer, or UNION)
  - Aggregate functions with or without GROUP BY or HAVING clauses
  - Base tables or views that are not updatable
  - Subqueries
  - ORDER BY clauses not based on an index
- Queries against a remote database server are parsed by the server. If the *RequestLive* property is set to *True*, the SQL statement must abide by Local SQL standards in addition to any server-imposed restrictions because the BDE needs to use it for conveying data changes to the table. A live result set for a query against a single table or view is returned if the query does not contain any of the following:
  - A DISTINCT clause in the SELECT statement
  - Aggregate functions, with or without GROUP BY or HAVING clauses
  - References to more than one base table or updatable views (joins)
  - Subqueries that reference the table in the FROM clause or other tables

If an application requests and receives a live result set, the *CanModify* property of the query component is set to *True*. Even if the query returns a live result set, you may not be able to update the result set directly if it contains linked fields or you switch indexes before attempting an update. If these conditions exist, you should treat the result set as a read-only result set, and update it accordingly.

If an application requests a live result set, but the SELECT statement syntax does not allow it, the BDE returns either

- A read-only result set for queries made against Paradox or dBASE.
- An error code for SQL queries made against a remote server.

### Updating read-only result sets

Applications can update data returned in a read-only result set if they are using cached updates.

If you are using a client dataset to cache updates, the client dataset or its associated provider can automatically generate the SQL for applying updates unless the query represents multiple tables. If the query represents multiple tables, you must indicate how to apply the updates:

- If all updates are applied to a single database table, you can indicate the underlying table to update in an *OnGetTableName* event handler.
- If you need more control over applying updates, you can associate the query with an update object (*TUpdateSQL*). A provider automatically uses this update object to apply updates:
  - **a** Associate the update object with the query by setting the query's *UpdateObject* property to the *TUpdateSQL* object you are using.
  - **b** Set the update object's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties to SQL statements that perform the appropriate updates for your query's data.

If you are using the BDE to cache updates, you must use an update object.

**Note** For more information on using update objects, see "Using update objects to update a dataset" on page 26-40.

## Using TStoredProc

*TStoredProc* represents a stored procedure. It implements all of the basic functionality introduced by *TDataSet*, as well as most of the special features typical of stored procedure-type datasets. Before looking at the unique features introduced by *TStoredProc*, you should familiarize yourself with the common database features described in "Understanding datasets," including the section on stored procedure-type datasets that starts on page 24-50.

Because *TStoredProc* is a BDE-enabled dataset, it must be associated with a database and a session. "Associating a dataset with database and session connections" on page 26-3 describes how you form these associations. Once the dataset is associated with a database and session, you can bind it to a particular stored procedure by setting the *StoredProcName* property.

*TStoredProc* differs from other stored procedure-type datasets in the following ways:

- It gives you greater control over how to bind parameters.
- It provides support for Oracle overloaded stored procedures.

### Binding parameters

When you prepare and execute a stored procedure, its input parameters are automatically bound to parameters on the server.

*TStoredProc* lets you use the *ParamBindMode* property to specify how parameters should be bound to the parameters on the server. By default *ParamBindMode* is set to *pbByName*, meaning that parameters from the stored procedure component are matched to those on the server by name. This is the easiest method of binding parameters.

Some servers also support binding parameters by ordinal value, the order in which the parameters appear in the stored procedure. In this case the order in which you specify parameters in the parameter collection editor is significant. The first parameter you specify is matched to the first input parameter on the server, the second parameter is matched to the second input parameter on the server, and so on. If your server supports parameter binding by ordinal value, you can set *ParamBindMode* to *pbByNumber*.

**Tip** If you want to set *ParamBindMode* to *pbByNumber*, you need to specify the correct parameter types in the correct order. You can view a server's stored procedure source code in the SQL Explorer to determine the correct order and type of parameters to specify.

### Working with Oracle overloaded stored procedures

Oracle servers allow overloading of stored procedures; overloaded procedures are different procedures with the same name. The stored procedure component's *Overload* property enables an application to specify the procedure to execute.

If *Overload* is zero (the default), there is assumed to be no overloading. If *Overload* is one (1), then the stored procedure component executes the first stored procedure it finds on the Oracle server that has the overloaded name; if it is two (2), it executes the second, and so on.

**Note** Overloaded stored procedures may take different input and output parameters. See your Oracle server documentation for more information.

## Connecting to databases with TDatabase

When a Delphi application uses the Borland Database Engine (BDE) to connect to a database, that connection is encapsulated by a *TDatabase* component. A database component represents the connection to a single database in the context of a BDE session.

*TDatabase* performs many of the same tasks as and shares many common properties, methods, and events with other database connection components. These commonalities are described in Chapter 23, "Connecting to databases."

In addition to the common properties, methods, and events, *TDatabase* introduces many BDE-specific features. These features are described in the following topics.

### Associating a database component with a session

All database components must be associated with a BDE session. Use the *SessionName*, establish this association. When you first create a database component at design time, *SessionName* is set to “Default”, meaning that it is associated with the default session component that is referenced by the global *Session* variable.

Multi-threaded or reentrant BDE applications may require more than one session. If you need to use multiple sessions, add *TSession* components for each session. Then, associate your dataset with a session component by setting the *SessionName* property to a session component’s *SessionName* property.

At runtime, you can access the session component with which the database is associated by reading the *Session* property. If *SessionName* is blank or “Default”, then the *Session* property references the same *TSession* instance referenced by the global *Session* variable. *Session* enables applications to access the properties, methods, and events of a database component’s parent session component without knowing the session’s actual name.

For more information about BDE sessions, see “Managing database sessions” on page 26-16.

If you are using an implicit database component, the session for that database component is the one specified by the dataset’s *SessionName* property.

### Understanding database and session component interactions

In general, session component properties provide global, default behaviors that apply to all implicit database components created at runtime. For example, the controlling session’s *KeepConnections* property determines whether a database connection is maintained even if its associated datasets are closed (the default), or if the connections are dropped when all its datasets are closed. Similarly, the default *OnPassword* event for a session guarantees that when an application attempts to attach to a database on a server that requires a password, it displays a standard password prompt dialog box.

Session methods apply somewhat differently. *TSession* methods affect all database components, regardless of whether they are explicitly created or instantiated implicitly by a dataset. For example, the session method *DropConnections* closes all datasets belonging to a session’s database components, and then drops all database connections, even if the *KeepConnection* property for individual database components is *True*.

Database component methods apply only to the datasets associated with a given database component. For example, suppose the database component *Database1* is associated with the default session. *Database1.CloseDataSets()* closes only those datasets associated with *Database1*. Open datasets belonging to other database components within the default session remain open.

### Identifying the database

*AliasName* and *DriverName* are mutually exclusive properties that identify the database server to which the *TDatabase* component connects.

- *AliasName* specifies the name of an existing BDE alias to use for the database component. The alias appears in subsequent drop-down lists for dataset components so that you can link them to a particular database component. If you specify *AliasName* for a database component, any value already assigned to *DriverName* is cleared because a driver name is always part of a BDE alias.

  You create and edit BDE aliases using the Database Explorer or the BDE Administration utility. For more information about creating and maintaining BDE aliases, see the online documentation for these utilities.

- *DriverName* is the name of a BDE driver. A driver name is one parameter in a BDE alias, but you may specify a driver name instead of an alias when you create a local BDE alias for a database component using the *DatabaseName* property. If you specify *DriverName*, any value already assigned to *AliasName* is cleared to avoid potential conflicts between the driver name you specify and the driver name that is part of the BDE alias identified in *AliasName*.

*DatabaseName* lets you provide your own name for a database connection. The name you supply is in addition to *AliasName* or *DriverName*, and is local to your application. *DatabaseName* can be a BDE alias, or, for Paradox and dBASE files, a fully-qualified path name. Like *AliasName*, *DatabaseName* appears in subsequent drop-down lists for dataset components to let you link them to database components.

At design time, to specify a BDE alias, assign a BDE driver, or create a local BDE alias, double-click a database component to invoke the Database Properties editor.

You can enter a *DatabaseName* in the Name edit box in the properties editor. You can enter an existing BDE alias name in the Alias name combo box for the *Alias* property, or you can choose from existing aliases in the drop-down list. The Driver name combo box enables you to enter the name of an existing BDE driver for the *DriverName* property, or you can choose from existing driver names in the drop-down list.

**Note** The Database Properties editor also lets you view and set BDE connection parameters, and set the states of the *LoginPrompt* and *KeepConnection* properties. For information on connection parameters, see "Setting BDE alias parameters" below. For information on *LoginPrompt*, see "Controlling server login" on page 23-4. For information on *KeepConnection* see "Opening a connection using TDatabase" on page 26-15.

#### Setting BDE alias parameters

At design time you can create or edit connection parameters in three ways:

- Use the Database Explorer or BDE Administration utility to create or modify BDE aliases, including parameters. For more information about these utilities, see their online Help files.

- Double-click the *Params* property in the Object Inspector to invoke the String List editor.
- Double-click a database component in a data module or form to invoke the Database Properties editor.

All of these methods edit the *Params* property for the database component. *Params* is a string list containing the database connection parameters for the BDE alias associated with a database component. Some typical connection parameters include path statement, server name, schema caching size, language driver, and SQL query mode.

When you first invoke the Database Properties editor, the parameters for the BDE alias are not visible. To see the current settings, click Defaults. The current parameters are displayed in the Parameter overrides memo box. You can edit existing entries or add new ones. To clear existing parameters, click Clear. Changes you make take effect only when you click OK.

At runtime, an application can set alias parameters only by editing the *Params* property directly. For more information about parameters specific to using SQL Links drivers with the BDE, see your online SQL Links help file.

## Opening a connection using TDatabase

As with all database connection components, to connect to a database using *TDatabase*, you set the *Connected* property to *True* or call the *Open* method. This process is described in "Connecting to a database server" on page 23-3. Once a database connection is established the connection is maintained as long as there is at least one active dataset. When there are no more active datasets, the connection is dropped unless the database component's *KeepConnection* property is *True*.

When you connect to a remote database server from an application, the application uses the BDE and the Borland SQL Links driver to establish the connection. (The BDE can also communicate with an ODBC driver that you supply.) You need to configure the SQL Links or ODBC driver for your application prior to making the connection. SQL Links and ODBC parameters are stored in the *Params* property of a database component. For information about SQL Links parameters, see the online *SQL Links User's Guide*. To edit the *Params* property, see "Setting BDE alias parameters" on page 26-14.

### Working with network protocols

As part of configuring the appropriate SQL Links or ODBC driver, you may need to specify the network protocol used by the server, such as SPX/IPX or TCP/IP, depending on the driver's configuration options. In most cases, network protocol configuration is handled using a server's client setup software. For ODBC it may also be necessary to check the driver setup using the ODBC driver manager.

Establishing an initial connection between client and server can be problematic. The following troubleshooting checklist should be helpful if you encounter difficulties:

- Is your server's client-side connection properly configured?
- Are the DLLs for your connection and database drivers in the search path?

- If you are using TCP/IP:
  - Is your TCP/IP communications software installed? Is the proper WINSOCK.DLL installed?
  - Is the server's IP address registered in the client's HOSTS file?
  - Is the Domain Name Services (DNS) properly configured?
  - Can you ping the server?

For more troubleshooting information, see the online *SQL Links User's Guide* and your server documentation.

#### Using ODBC

An application can use ODBC data sources (for example, Btrieve). An ODBC driver connection requires

- A vendor-supplied ODBC driver.
- The Microsoft ODBC Driver Manager.
- The BDE Administration utility.

To set up a BDE alias for an ODBC driver connection, use the BDE Administration utility. For more information, see the BDE Administration utility's online help file.

### Using database components in data modules

You can safely place database components in data modules. If you put a data module that contains a database component into the Object Repository, however, and you want other users to be able to inherit from it, you must set the *HandleShared* property of the database component to *True* to prevent global name space conflicts.

## Managing database sessions

An BDE-based application's database connections, drivers, cursors, queries, and so on are maintained within the context of one or more BDE sessions. Sessions isolate a set of database access operations, such as database connections, without the need to start another instance of the application.

All BDE-based database applications automatically include a default session component, named *Session*, that encapsulates the default BDE session. When database components are added to the application, they are automatically associated with the default session (note that its *SessionName* is "Default"). The default session provides global control over all database components not associated with another session, whether they are implicit (created by the session at runtime when you open a dataset that is not associated with a database component you create) or persistent (explicitly created by your application). The default session is not visible in your data module or form at design time, but you can access its properties and methods in your code at runtime.

To use the default session, you need write no code unless your application must

- Explicitly activate or deactivate a session, enabling or disabling the session's databases' ability to open.
- Modify the properties of the session, such as specifying default properties for implicitly generated database components.
- Execute a session's methods, such as managing database connections (for example opening and closing database connections in response to user actions).
- Respond to session events, such as when the application attempts to access a password-protected Paradox or dBASE table.
- Set Paradox directory locations such as the *NetFileDir* property to access Paradox tables on a network and the *PrivateDir* property to a local hard drive to speed performance.
- Manage the BDE aliases that describe possible database connection configurations for databases and datasets that use the session.

Whether you add database components to an application at design time or create them dynamically at runtime, they are automatically associated with the default session unless you specifically assign them to a different session. If you open a dataset that is not associated with a database component, Delphi automatically

- Creates a database component for it at runtime.
- Associates the database component with the default session.
- Initializes some of the database component's key properties based on the default session's properties. Among the most important of these properties is *KeepConnections*, which determines when database connections are maintained or dropped by an application.

The default session provides a widely applicable set of defaults that can be used as is by most applications. You need only associate a database component with an explicitly named session if the component performs a simultaneous query against a database already opened by the default session. In this case, each concurrent query must run under its own session. Multi-threaded database applications also require multiple sessions, where each thread has its own session.

Applications can create additional session components as needed. BDE-based database applications automatically include a session list component, named *Sessions*, that you can use to manage all of your session components. For more information about managing multiple sessions see, "Managing multiple sessions" on page 26-29.

You can safely place session components in data modules. If you put a data module that contains one or more session components into the Object Repository, however, make sure to set the *AutoSessionName* property to *True* to avoid namespace conflicts when users inherit from it.

### Activating a session

*Active* is a Boolean property that determines if database and dataset components associated with a session are open. You can use this property to read the current state of a session's database and dataset connections, or to change it. If *Active* is *False* (the default), all databases and datasets associated with the session are closed. If *True*, databases and datasets are open.

A session is activated when it is first created, and subsequently, whenever its *Active* property is changed to *True* from *False* (for example, when a database or dataset is associated with a session is opened and there are currently no other open databases or datasets). Setting *Active* to *True* triggers a session's *OnStartup* event, registers the paradox directory locations with the BDE, and registers the *ConfigMode* property, which determines what BDE aliases are available within the session. You can write an *OnStartup* event handler to initialize the *NetFileDir*, *PrivateDir*, and *ConfigMode* properties before they are registered with the BDE, or to perform other specific session start-up activities. For information about the *NetFileDir* and *PrivateDir* properties, see "Specifying Paradox directory locations" on page 26-24. For information about ConfigMode, see "Working with BDE aliases" on page 26-25.

Once a session is active, you can open its database connections by calling the *OpenDatabase* method.

For session components you place in a data module or form, setting *Active* to *False* when there are open databases or datasets closes them. At runtime, closing databases and datasets may trigger events associated with them.

**Note** You cannot set *Active* to *False* for the default session at design time. While you can close the default session at runtime, it is not recommended.

You can also use a session's *Open* and *Close* methods to activate or deactivate sessions other than the default session at runtime. For example, the following single line of code closes all open databases and datasets for a session:

```
Session1.Close;
```

This code sets Session1's *Active* property to *False*. When a session's *Active* property is *False*, any subsequent attempt by the application to open a database or dataset resets *Active* to *True* and calls the session's *OnStartup* event handler if it exists. You can also explicitly code session reactivation at runtime. The following code reactivates *Session1*:

```
Session1.Open;
```

**Note** If a session is active you can also open and close individual database connections. For more information, see "Closing database connections" on page 26-20.

### Specifying default database connection behavior

*KeepConnections* provides the default value for the *KeepConnection* property of implicit database components created at runtime. *KeepConnection* specifies what happens to a database connection established for a database component when all its datasets are closed. If *True* (the default), a constant, or *persistent*, database connection is maintained even if no dataset is active. If *False*, a database connection is dropped as soon as all its datasets are closed.

**Note** Connection persistence for a database component you explicitly place in a data module or form is controlled by that database component's *KeepConnection* property. If set differently, *KeepConnection* for a database component always overrides the *KeepConnections* property of the session. For more information about controlling individual database connections within a session, see "Managing database connections" on page 26-19.

*KeepConnections* should be set to *True* for applications that frequently open and close all datasets associated with a database on a remote server. This setting reduces network traffic and speeds data access because it means that a connection need only be opened and closed once during the lifetime of the session. Otherwise, every time the application closes or reestablishes a connection, it incurs the overhead of attaching and detaching the database.

**Note** Even when *KeepConnections* is *True* for a session, you can close and free inactive database connections for all implicit database components by calling the *DropConnections* method. For more information about *DropConnections*, see "Dropping inactive database connections" on page 26-20.

## Managing database connections

You can use a session component to manage the database connections within it. The session component includes properties and methods you can use to

- Open database connections.
- Close database connections.
- Close and free all inactive temporary database connections.
- Locate specific database connections.
- Iterate through all open database connections.

### Opening database connections

To open a database connection within a session, call the *OpenDatabase* method. *OpenDatabase* takes one parameter, the name of the database to open. This name is a BDE alias or the name of a database component. For Paradox or dBASE, the name can also be a fully qualified path name. For example, the following statement uses the default session and attempts to open a database connection for the database pointed to by the DBDEMOS alias:

```
var
  DBDemosDatabase: TDatabase;
begin
  DBDemosDatabase := Session.OpenDatabase('DBDEMOS');
  ...
```

*OpenDatabase* actives the session if it is not already active, and then checks if the specified database name matches the *DatabaseName* property of any database components for the session. If the name does not match an existing database component, *OpenDatabase* creates a temporary database component using the specified name. Finally, *OpenDatabase* calls the *Open* method of the database component to connect to the server. Each call to *OpenDatabase* increments a reference count for the database by 1. As long as this reference count remains greater than 0, the database is open.

### Closing database connections

To close an individual database connection, call the *CloseDatabase* method. When you call *CloseDatabase*, the reference count for the database, which is incremented when you call *OpenDatabase*, is decremented by 1. When the reference count for a database is 0, the database is closed. *CloseDatabase* takes one parameter, the database to close. If you opened the database using the *OpenDatabase* method, this parameter can be set to the return value of *OpenDatabase*.

```
Session.CloseDatabase(DBDemosDatabase);
```

If the specified database name is associated with a temporary (implicit) database component, and the session's *KeepConnections* property is *False*, the database component is freed, effectively closing the connection.

**Note** If *KeepConnections* is *False* temporary database components are closed and freed automatically when the last dataset associated with the database component is closed. An application can always call *CloseDatabase* prior to that time to force closure. To free temporary database components when *KeepConnections* is *True*, call the database component's *Close* method, and then call the session's *DropConnections* method.

**Note** Calling *CloseDatabase* for a persistent database component does not actually close the connection. To close the connection, call the database component's *Close* method directly.

There are two ways to close all database connections within the session:

- Set the *Active* property for the session to *False*.
- Call the *Close* method for the session.

When you set *Active* to *False*, Delphi automatically calls the *Close* method. *Close* disconnects from all active databases by freeing temporary database components and calling each persistent database component's *Close* method. Finally, *Close* sets the session's BDE handle to **nil**.

### Dropping inactive database connections

If the *KeepConnections* property for a session is *True* (the default), then database connections for temporary database components are maintained even if all the datasets used by the component are closed. You can eliminate these connections and free all inactive temporary database components for a session by calling the *DropConnections* method. For example, the following code frees all inactive, temporary database components for the default session:

```
Session.DropConnections;
```

Temporary database components for which one or more datasets are active are not dropped or freed by this call. To free these components, call *Close*.

### Searching for a database connection

Use a session's *FindDatabase* method to determine whether a specified database component is already associated with a session. *FindDatabase* takes one parameter, the name of the database to search for. This name is a BDE alias or database component name. For Paradox or dBASE, it can also be a fully-qualified path name.

*FindDatabase* returns the database component if it finds a match. Otherwise it returns **nil**.

The following code searches the default session for a database component using the DBDEMOS alias, and if it is not found, creates one and opens it:

```
var
  DB: TDatabase;
begin
  DB := Session.FindDatabase('DBDEMOS');
  if (DB = nil) then                           { database doesn't exist for session so,}
    DB := Session.OpenDatabase('DBDEMOS');     { create and open it}
  if Assigned(DB) and DB.Connected then begin
    DB.StartTransaction;
    ...
  end;
end;
```

### Iterating through a session's database components

You can use two session component properties, *Databases* and *DatabaseCount*, to cycle through all the active database components associated with a session.

*Databases* is an array of all currently active database components associated with a session. *DatabaseCount* is the number of databases in that array. As connections are opened or closed during a session's life-span, the values of *Databases* and *DatabaseCount* change. For example, if a session's *KeepConnections* property is *False* and all database components are created as needed at runtime, each time a unique database is opened, *DatabaseCount* increases by one. Each time a unique database is closed, *DatabaseCount* decreases by one. If *DatabaseCount* is zero, there are no currently active database components for the session.

The following example code sets the *KeepConnection* property of each active database in the default session to *True*:

```
var
  MaxDbCount: Integer;
begin
  with Session do
    if (DatabaseCount > 0) then
      for MaxDbCount := 0 to (DatabaseCount - 1) do
        Databases[MaxDbCount].KeepConnection := True;
end;
```

## Working with password-protected Paradox and dBASE tables

A session component can store passwords for password-protected Paradox and dBASE tables. Once you add a password to the session, your application can open tables protected by that password. Once you remove the password from the session, your application can't open tables that use the password until you add it again.

### Using the AddPassword method

The *AddPassword* method provides an optional way for an application to provide a password for a session prior to opening an encrypted Paradox or dBASE table that requires a password for access. If you do not add the password to the session, when your application attempts to open a password-protected table, a dialog box prompts the user for a password.

*AddPassword* takes one parameter, a string containing the password to use. You can call *AddPassword* as many times as necessary to add passwords (one at a time) to access tables protected with different passwords.

```
var
  Passwrd: String;
begin
  Passwrd := InputBox('Enter password', 'Password:', '');
  Session.AddPassword(Passwrd);
  try
    Table1.Open;
  except
    ShowMessage('Could not open table!');
    Application.Terminate;
  end;
end;
```

**Note** Use of the *InputBox* function, above, is for demonstration purposes. In a real-world application, use password entry facilities that mask the password as it is entered, such as the *PasswordDialog* function or a custom form.

The Add button of the *PasswordDialog* function dialog has the same effect as the *AddPassword* method.

```
if PasswordDialog(Session) then
  Table1.Open
else
  ShowMessage('No password given, could not open table!');
end;
```

### Using the RemovePassword and RemoveAllPasswords methods

*RemovePassword* deletes a previously added password from memory. *RemovePassword* takes one parameter, a string containing the password to delete.

```
Session.RemovePassword(‘secret’);
```

*RemoveAllPasswords* deletes all previously added passwords from memory.

```
Session.RemoveAllPasswords;
```

### Using the GetPassword method and OnPassword event

The *OnPassword* event allows you to control how your application supplies passwords for Paradox and dBASE tables when they are required. Provide a handler for the *OnPassword* event if you want to override the default password handling behavior. If you do not provide a handler, Delphi presents a default dialog for entering a password and no special behavior is provided—the table open attempt either succeeds or an exception is raised.

If you provide a handler for the *OnPassword* event, do two things in the event handler: call the *AddPassword* method and set the event handler's *Continue* parameter to *True*. The *AddPassword* method passes a string to the session to be used as a password for the table. The *Continue* parameter indicates to Delphi that no further password prompting need be done for this table open attempt. The default value for *Continue* is *False*, and so requires explicitly setting it to *True*. If *Continue* is *False* after the event handler has finished executing, an *OnPassword* event fires again—even if a valid password has been passed using *AddPassword*. If *Continue* is *True* after execution of the event handler and the string passed with *AddPassword* is not the valid password, the table open attempt fails and an exception is raised.

*OnPassword* can be triggered by two circumstances. The first is an attempt to open a password-protected table (dBASE or Paradox) when a valid password has not already been supplied to the session. (If a valid password for that table has already been supplied, the *OnPassword* event does not occur.)

The other circumstance is a call to the *GetPassword method. GetPassword either generates an OnPassword event, or, if the session does not have an OnPassword event handler, displays a default password dialog. It returns True*e *if the OnPassword event handler or default dialog added a password to the session, and False if no entry at all was made.*

In the following example, the *Password* method is designated as the *OnPassword* event handler for the default session by assigning it to the global *Session* object's *OnPassword* property.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Session.OnPassword := Password;
end;
```

In the *Password* method, the *InputBox* function prompts the user for a password. The *AddPassword* method then programmatically supplies the password entered in the dialog to the session.

```
procedure TForm1.Password(Sender: TObject; var Continue: Boolean);
var
  Passwrd: String;
begin
  Passwrd := InputBox('Enter password', 'Password:', '');
  Continue := (Passwrd > '');
  Session.AddPassword(Passwrd);
end;
```

The *OnPassword* event (and thus the *Password* event handler) is triggered by an attempt to open a password-protected table, as demonstrated below. Even though the user is prompted for a password in the handler for the *OnPassword* event, the table open attempt can still fail if they enter an invalid password or something else goes wrong.

```
procedure TForm1.OpenTableBtnClick(Sender: TObject);
const
  CRLF = #13 + #10;
```

```
begin
  try
    Table1.Open;                              { this line triggers the OnPassword event }
  except
    on E:Exception do begin                        { exception if cannot open table }
      ShowMessage('Error!' + CRLF +           { display error explaining what happened }
        E.Message + CRLF +
        'Terminating application...');
      Application.Terminate;                                    { end the application }
    end;
  end;
end;
```

## Specifying Paradox directory locations

Two session component properties, *NetFileDir* and *PrivateDir*, are specific to applications that work with Paradox tables.

*NetFileDir* specifies the directory that contains the Paradox network control file, PDOXUSRS.NET. This file governs sharing of Paradox tables on network drives. All applications that need to share Paradox tables must specify the same directory for the network control file (typically a directory on a network file server). Delphi derives a value for *NetFileDir* from the Borland Database Engine (BDE) configuration file for a given database alias. If you set *NetFileDir* yourself, the value you supply overrides the BDE configuration setting, so be sure to validate the new value.

At design time, you can specify a value for *NetFileDir* in the Object Inspector. You can also set or change *NetFileDir* in code at runtime. The following code sets *NetFileDir* for the default session to the location of the directory from which your application runs:

```
Session.NetFileDir := ExtractFilePath(Application.EXEName);
```

**Note** *NetFileDir* can only be changed when an application does not have any open Paradox files. If you change *NetFileDir* at runtime, verify that it points to a valid network directory that is shared by your network users.

*PrivateDir* specifies the directory for storing temporary table processing files, such as those generated by the BDE to handle local SQL statements. If no value is specified for the *PrivateDir* property, the BDE automatically uses the current directory at the time it is initialized. If your application runs directly from a network file server, you can improve application performance at runtime by setting *PrivateDir* to a user's local hard drive before opening the database.

**Note** Do not set *PrivateDir* at design time and then open the session in the IDE. Doing so generates a Directory is busy error when running your application from the IDE.

The following code changes the setting of the default session's *PrivateDir* property to a user's C:\TEMP directory:

```
Session.PrivateDir := 'C:\TEMP';
```

**Important** Do not set *PrivateDir* to a root directory on a drive. Always specify a subdirectory.

## Working with BDE aliases

Each database component associated with a session has a BDE alias (although optionally a fully-qualified path name may be substituted for an alias when accessing Paradox and dBASE tables). A session can create, modify, and delete aliases during its lifetime.

The *AddAlias* method creates a new BDE alias for an SQL database server. *AddAlias* takes three parameters: a string containing a name for the alias, a string that specifies the SQL Links driver to use, and a string list populated with parameters for the alias. For example, the following statements use *AddAlias* to add a new alias for accessing an InterBase server to the default session:

```
var
  AliasParams: TStringList;
begin
  AliasParams := TStringList.Create;
  try
    with AliasParams do begin
      Add('OPEN MODE=READ');
      Add('USER NAME=TOMSTOPPARD');
      Add('SERVER NAME=ANIMALS:/CATS/PEDIGREE.GDB');
    end;
    Session.AddAlias('CATS', 'INTRBASE', AliasParams);
    ...
  finally
    AliasParams.Free;
  end;
end;
```

*AddStandardAlias* creates a new BDE alias for Paradox, dBASE, or ASCII tables. *AddStandardAlias* takes three string parameters: the name for the alias, the fully-qualified path to the Paradox or dBASE table to access, and the name of the default driver to use when attempting to open a table that does not have an extension. For example, the following statement uses *AddStandardAlias* to create a new alias for accessing a Paradox table:

```
AddStandardAlias('MYDBDEMOS', 'C:\TESTING\DEMOS\', 'Paradox');
```

When you add an alias to a session, the BDE stores a copy of the alias in memory, where it is only available to this session and any other sessions with *cfmPersistent* included in the *ConfigMode* property. *ConfigMode* is a set that describes which types of aliases can be used by the databases in the session. The default setting is *cmAll*, which translates into the set [*cfmVirtual*, *cfmPersistent*, *cfmSession*]. If *ConfigMode* is *cmAll*, a session can see all aliases created within the session (*cfmSession*), all aliases in the BDE configuration file on a user's system (*cfmPersistent*), and all aliases that the BDE maintains in memory (*cfmVirtual*). You can change *ConfigMode* to restrict what BDE aliases the databases in a session can use. For example, setting *ConfigMode* to *cfmSession* restricts a session's view of aliases to those created within the session. All other aliases in the BDE configuration file and in memory are not available.

To make a newly created alias available to all sessions and to other applications, use the session's *SaveConfigFile* method. *SaveConfigFile* writes aliases in memory to the BDE configuration file where they can be read and used by other BDE-enabled applications.

After you create an alias, you can make changes to its parameters by calling *ModifyAlias*. *ModifyAlias* takes two parameters: the name of the alias to modify and a string list containing the parameters to change and their values. For example, the following statements use *ModifyAlias* to change the OPEN MODE parameter for the CATS alias to READ/WRITE in the default session:

```
var
  List: TStringList;
begin
  List := TStringList.Create;
  with List do begin
    Clear;
    Add('OPEN MODE=READ/WRITE');
  end;
  Session.ModifyAlias('CATS', List);
  List.Free;
  ...
```

To delete an alias previously created in a session, call the *DeleteAlias* method. *DeleteAlias* takes one parameter, the name of the alias to delete. *DeleteAlias* makes an alias unavailable to the session.

**Note** *DeleteAlias* does not remove an alias from the BDE configuration file if the alias was written to the file by a previous call to *SaveConfigFile*. To remove the alias from the configuration file after calling *DeleteAlias*, call *SaveConfigFile* again.

Session components provide five methods for retrieving information about a BDE aliases, including parameter information and driver information. They are:

- *GetAliasNames*, to list the aliases to which a session has access.
- *GetAliasParams*, to list the parameters for a specified alias.
- *GetAliasDriverName*, to return the name of the BDE driver used by the alias.
- *GetDriverNames*, to return a list of all BDE drivers available to the session.
- *GetDriverParams*, to return driver parameters for a specified driver.

For more information about using a session's informational methods, see "Retrieving information about a session" below. For more information about BDE aliases and the SQL Links drivers with which they work, see the BDE online help, BDE32.HLP.

## Retrieving information about a session

You can retrieve information about a session and its database components by using a session's informational methods. For example, one method retrieves the names of all aliases known to the session, and another method retrieves the names of tables associated with a specific database component used by the session. Table 26.4 summarizes the informational methods to a session component:

**Table 26.4** Database-related informational methods for session components

| Method | Purpose |
|---|---|
| GetAliasDriverName | Retrieves the BDE driver for a specified alias of a database. |
| GetAliasNames | Retrieves the list of BDE aliases for a database. |
| GetAliasParams | Retrieves the list of parameters for a specified BDE alias of a database. |
| GetConfigParams | Retrieves configuration information from the BDE configuration file. |
| GetDatabaseNames | Retrieves the list of BDE aliases and the names of any *TDatabase* components currently in use. |
| GetDriverNames | Retrieves the names of all currently installed BDE drivers. |
| GetDriverParams | Retrieves the list of parameters for a specified BDE driver. |
| GetStoredProcNames | Retrieves the names of all stored procedures for a specified database. |
| GetTableNames | Retrieves the names of all tables matching a specified pattern for a specified database. |
| GetFieldNames | Retrieves the names of all fields in a specified table in a specified database. |

Except for *GetAliasDriverName*, these methods return a set of values into a string list declared and maintained by your application. (*GetAliasDriverName* returns a single string, the name of the current BDE driver for a particular database component used by the session.)

For example, the following code retrieves the names of all database components and aliases known to the default session:

```
var
  List: TStringList;
begin
  List := TStringList.Create;
  try
    Session.GetDatabaseNames(List);
    ...
  finally
    List.Free;
  end;
end;
```

## Creating additional sessions

You can create sessions to supplement the default session. At design time, you can place additional sessions on a data module (or form), set their properties in the Object Inspector, write event handlers for them, and write code that calls their methods. You can also create sessions, set their properties, and call their methods at runtime.

**Note** Creating additional sessions is optional unless an application runs concurrent queries against a database or the application is multi-threaded.

To enable dynamic creation of a session component at runtime, follow these steps:

1. Declare a *TSession* variable.
2. Instantiate a new session by calling the *Create* method. The constructor sets up an empty list of database components for the session, sets the *KeepConnections* property to *True*, and adds the session to the list of sessions maintained by the application's session list component.
3. Set the *SessionName* property for the new session to a unique name. This property is used to associate database components with the session. For more information about the *SessionName* property, see "Naming a session" on page 26-29.
4. Activate the session and optionally adjust its properties.

You can also create and open sessions using the *OpenSession* method of *TSessionList*. Using *OpenSession* is safer than calling *Create*, because *OpenSession* only creates a session if it does not already exist. For information about *OpenSession*, see "Managing multiple sessions" on page 26-29.

The following code creates a new session component, assigns it a name, and opens the session for database operations that follow (not shown here). After use, it is destroyed with a call to the *Free* method.

**Note** Never delete the default session.

```
var
  SecondSession: TSession;
begin
  SecondSession := TSession.Create(Form1);
  with SecondSession do
    try
      SessionName := 'SecondSession';
      KeepConnections := False;
      Open;
      ...
    finally
      SecondSession.Free;
    end;
end;
```

### Naming a session

A session's *SessionName* property is used to name the session so that you can associate databases and datasets with it. For the default session, *SessionName* is "Default," For each additional session component you create, you must set its *SessionName* property to a unique value.

Database and dataset components have *SessionName* properties that correspond to the *SessionName* property of a session component. If you leave the *SessionName* property blank for a database or dataset component it is automatically associated with the default session. You can also set *SessionName* for a database or dataset component to a name that corresponds to the *SessionName* of a session component you create.

The following code uses the *OpenSession* method of the default *TSessionList* component, *Sessions*, to open a new session component, sets its *SessionName* to "InterBaseSession," activate the session, and associate an existing database component *Database1* with that session:

```
var
  IBSession: TSession;
  ƒ
begin
  IBSession := Sessions.OpenSession('InterBaseSession');
  Database1.SessionName := 'InterBaseSession';
end;
```

### Managing multiple sessions

If you create a single application that uses multiple threads to perform database operations, you must create one additional session for each thread. The BDE page on the Component palette contains a session component that you can place in a data module or on a form at design time.

**Important** When you place a session component, you must also set its *SessionName* property to a unique value so that it does not conflict with the default session's *SessionName* property.

Placing a session component at design time presupposes that the number of threads (and therefore sessions) required by the application at runtime is static. More likely, however, is that an application needs to create sessions dynamically. To create sessions dynamically, call the *OpenSession* method of the global *Sessions* object at runtime.

*OpenSession* requires a single parameter, a name for the session that is unique across all session names for the application. The following code dynamically creates and activates a new session with a uniquely generated name:

```
Sessions.OpenSession('RunTimeSession' + IntToStr(Sessions.Count + 1));
```

This statement generates a unique name for a new session by retrieving the current number of sessions, and adding one to that value. Note that if you dynamically create and destroy sessions at runtime, this example code will not work as expected. Nevertheless, this example illustrates how to use the properties and methods of *Sessions* to manage multiple sessions.

*Sessions* is a variable of type *TSessionList* that is automatically instantiated for BDE-based database applications. You use the properties and methods of *Sessions* to keep track of multiple sessions in a multi-threaded database application. Table 26.5 summarizes the properties and methods of the *TSessionList* component:

**Table 26.5** TSessionList properties and methods

| Property or Method | Purpose |
|---|---|
| *Count* | Returns the number of sessions, both active and inactive, in the session list. |
| *FindSession* | Searches for a session with a specified name and returns a pointer to it, or **nil** if there is no session with the specified name. If passed a blank session name, *FindSession* returns a pointer to the default session, *Session*. |
| *GetSessionNames* | Populates a string list with the names of all currently instantiated session components. This procedure always adds at least one string, “Default” for the default session. |
| *List* | Returns the session component for a specified session name. If there is no session with the specified name, an exception is raised. |
| *OpenSession* | Creates and activates a new session or reactivates an existing session for a specified session name. |
| *Sessions* | Accesses the session list by ordinal value. |

As an example of using *Sessions* properties and methods in a multi-threaded application, consider what happens when you want to open a database connection. To determine if a connection already exists, use the *Sessions* property to walk through each session in the sessions list, starting with the default session. For each session component, examine its *Databases* property to see if the database in question is open. If you discover that another thread is already using the desired database, examine the next session in the list.

If an existing thread is not using the database, then you can open the connection within that session.

If, on the other hand, all existing threads are using the database, you must open a new session in which to open another database connection.

If you are replicating a data module that contains a session in a multi-threaded application, where each thread contains its own copy of the data module, you can use the *AutoSessionName* property to make sure that all datasets in the data module use the correct session. Setting *AutoSessionName* to *True* causes the session to generate its own unique name dynamically when it is created at runtime. It then assigns this name to every dataset in the data module, overriding any explicitly set session names. This ensures that each thread has its own session, and each dataset uses the session in its own data module.

## Using transactions with the BDE

By default, the BDE provides implicit transaction control for your applications. When an application is under implicit transaction control, a separate transaction is used for each record in a dataset that is written to the underlying database. Implicit transactions guarantee both a minimum of record update conflicts and a consistent view of the database. On the other hand, because each row of data written to a database takes place in its own transaction, implicit transaction control can lead to excessive network traffic and slower application performance. Also, implicit transaction control will not protect logical operations that span more than one record.

If you explicitly control transactions, you can choose the most effective times to start, commit, and roll back your transactions. When you develop applications in a multi-user environment, particularly when your applications run against a remote SQL server, you should control transactions explicitly.

There are two mutually exclusive ways to control transactions explicitly in a BDE-based database application:

- Use the database component to control transactions. The main advantage to using the methods and properties of a database component is that it provides a clean, portable application that is not dependent on a particular database or server. This type of transaction control is supported by all database connection components, and described in "Managing transactions" on page 23-6
- Use passthrough SQL in a query component to pass SQL statements directly to remote SQL or ODBC servers. The main advantage to passthrough SQL is that you can use the advanced transaction management capabilities of a particular database server, such as schema caching. To understand the advantages of your server's transaction management model, see your database server documentation. For more information about using passthrough SQL, see "Using passthrough SQL" below.

When working with local databases, you can only use the database component to create explicit transactions (local databases do not support passthrough SQL). However, there are limitations to using local transactions. For more information on using local transactions, see "Using local transactions" on page 26-32.

**Note** You can minimize the number of transactions you need by caching updates. For more information about cached updates, see "Using a client dataset to cache updates" and "Using the BDE to cache updates" on page 26-33.

## Using passthrough SQL

With passthrough SQL, you use a *TQuery*, *TStoredProc*, or *TUpdateSQL* component to send an SQL transaction control statement directly to a remote database server. The BDE does not process the SQL statement. Using passthrough SQL enables you to take direct advantage of the transaction controls offered by your server, especially when those controls are non-standard.

To use passthrough SQL to control a transaction, you must

- Install the proper SQL Links drivers. If you chose the "Typical" installation when installing Delphi, all SQL Links drivers are already properly installed.
- Configure your network protocol. See your network administrator for more information.
- Have access to a database on a remote server.
- Set SQLPASSTHRU MODE to NOT SHARED using the SQL Explorer. SQLPASSTHRU MODE specifies whether the BDE and passthrough SQL statements can share the same database connections. In most cases, SQLPASSTHRU MODE is set to SHARED AUTOCOMMIT. However, you can't share database connections when using transaction control statements. For more information about SQLPASSTHRU modes, see the help file for the BDE Administration utility.

**Note** When SQLPASSTHRU MODE is NOT SHARED, you must use separate database components for datasets that pass SQL transaction statements to the server and datasets that do not.

## Using local transactions

The BDE supports local transactions against Paradox, dBASE, Access, and FoxPro tables. From a coding perspective, there is no difference to you between a local transaction and a transaction against a remote database server.

**Note** When using transactions with local Paradox, dBASE, Access, and FoxPro tables, set *TransIsolation* to *tiDirtyRead* instead of using the default value of *tiReadCommitted*. A BDE error is returned if *TransIsolation* is set to anything but *tiDirtyRead* for local tables.

When a transaction is started against a local table, updates performed against the table are logged. Each log record contains the old record buffer for a record. When a transaction is active, records that are updated are locked until the transaction is committed or rolled back. On rollback, old record buffers are applied against updated records to restore them to their pre-update states.

Local transactions are more limited than transactions against SQL servers or ODBC drivers. In particular, the following limitations apply to local transactions:

- Automatic crash recovery is not provided.
- Data definition statements are not supported.

- Transactions cannot be run against temporary tables.
- *TransIsolation* level must only be set to *tiDirtyRead*.
- For Paradox, local transactions can only be performed on tables with valid indexes. Data cannot be rolled back on Paradox tables that do not have indexes.
- Only a limited number of records can be locked and modified. With Paradox tables, you are limited to 255 records. With dBASE the limit is 100.
- Transactions cannot be run against the BDE ASCII driver.
- Closing a cursor on a table during a transaction rolls back the transaction unless:
  - Several tables are open.
  - The cursor is closed on a table to which no changes were made.

## Using the BDE to cache updates

The recommended approach for caching updates is to use a client dataset (*TBDEClientDataSet*) or to connect the BDE-dataset to a client dataset using a dataset provider. The advantages of using a client dataset are discussed in "Using a client dataset to cache updates" on page 29-16.

For simple cases, however, you may choose to use the BDE to cache updates instead. BDE-enabled datasets and *TDatabase* components provide built-in properties, methods, and events for handling cached updates. Most of these correspond directly to the properties, methods, and events that you use with client datasets and dataset providers when using a client dataset to cache updates. The following table lists these properties, events, and methods and the corresponding properties, methods and events on *TBDEClientDataSet*:

**Table 26.6** Properties, methods, and events for cached updates

| On BDE-enabled datasets (or TDatabase) | On TBDEClientDataSet | Purpose |
|---|---|---|
| *CachedUpdates* | Not needed for client datasets, which always cache updates. | Determines whether cached updates are in effect for the dataset. |
| *UpdateObject* | Use a *BeforeUpdateRecord* event handler, or, if using *TClientDataSet*, use the *UpdateObject* property on the BDE-enabled source dataset. | Specifies the update object for updating read-only datasets. |
| *UpdatesPending* | *ChangeCount* | Indicates whether the local cache contains updated records that need to be applied to the database. |
| *UpdateRecordTypes* | *StatusFilter* | Indicates the kind of updated records to make visible when applying cached updates. |
| *UpdateStatus* | *UpdateStatus* | Indicates if a record is unchanged, modified, inserted, or deleted. |

**Table 26.6** Properties, methods, and events for cached updates (continued)

| On BDE-enabled datasets (or TDatabase) | On TBDEClientDataSet | Purpose |
|---|---|---|
| *OnUpdateError* | *OnReconcileError* | An event for handling update errors on a record-by-record basis. |
| *OnUpdateRecord* | *BeforeUpdateRecord* | An event for processing updates on a record-by-record basis. |
| *ApplyUpdates*<br>*ApplyUpdates* (database) | *ApplyUpdates* | Applies records in the local cache to the database. |
| *CancelUpdates* | *CancelUpdates* | Removes all pending updates from the local cache without applying them. |
| *CommitUpdates* | *Reconcile* | Clears the update cache following successful application of updates. |
| *FetchAll* | *GetNextPacket*<br>(and *PacketRecords*) | Copies database records to the local cache for editing and updating. |
| *RevertRecord* | *RevertRecord* | Undoes updates to the current record if updates are not yet applied. |

For an overview of the cached update process, see "Overview of using cached updates" on page 29-17.

**Note** Even if you are using a client dataset to cache updates, you may want to read the section about update objects on page 26-40. You can use update objects in the *BeforeUpdateRecord* event handler of *TBDEClientDataSet* or *TDataSetProvider* to apply updates from stored procedures or multi-table queries.

## Enabling BDE-based cached updates

To use the BDE for cached updates, the BDE-enabled dataset must indicate that it should cache updates. This is specified by setting the *CachedUpdates* property to *True*. When you enable cached updates, a copy of all records is cached in local memory. Users view and edit this local copy of data. Changes, insertions, and deletions are also cached in memory. They accumulate in memory until the application applies those changes to the database server. If changed records are successfully applied to the database, the record of those changes are freed in the cache.

The dataset caches all updates until you set *CachedUpdates* to *False*. Applying cached updates does not disable further cached updates; it only writes the current set of changes to the database and clears them from memory. Canceling the updates by calling *CancelUpdates* removes all the changes currently in the cache, but does not stop the dataset from caching any subsequent changes.

**Note** If you disable cached updates by setting *CachedUpdates* to *False*, any pending changes that you have not yet applied are discarded without notification. To prevent losing changes, test the *UpdatesPending* property before disabling cached updates.

## Applying BDE-based cached updates

Applying updates is a two-phase process that should occur in the context of a database component's transaction so that your application can recover gracefully from errors. For information about transaction handling with database components, see "Managing transactions" on page 23-6.

When applying updates under database transaction control, the following events take place:

1. A database transaction starts.
2. Cached updates are written to the database (phase 1). If you provide it, an *OnUpdateRecord* event is triggered once for each record written to the database. If an error occurs when a record is applied to the database, the *OnUpdateError* event is triggered if you provide one.
3. The transaction is committed if writes are successful or rolled back if they are not:

   If the database write is successful:

   - Database changes are committed, ending the database transaction.
   - Cached updates are committed, clearing the internal cache buffer (phase 2).

   If the database write is unsuccessful:

   - Database changes are rolled back, ending the database transaction.
   - Cached updates are not committed, remaining intact in the internal cache.

For information about creating and using an *OnUpdateRecord* event handler, see "Creating an OnUpdateRecord event handler" on page 26-37. For information about handling update errors that occur when applying cached updates, see "Handling cached update errors" on page 26-38.

**Note** Applying cached updates is particularly tricky when you are working with multiple datasets linked in a master/detail relationship because the order in which you apply updates to each dataset is significant. Usually, you must update master tables before detail tables, except when handling deleted records, where this order must be reversed. Because of this difficulty, it is strongly recommended that you use client datasets when caching updates in a master/detail form. Client datasets automatically handle all ordering issues with master/detail relationships.

There are two ways to apply BDE-based updates:

- You can apply updates using a database component by calling its *ApplyUpdates* method. This method is the simplest approach, because the database handles all details of managing a transaction for the update process and of clearing the dataset's cache when updating is complete.
- You can apply updates for a single dataset by calling the dataset's *ApplyUpdates* and *CommitUpdates* methods. When applying updates at the dataset level you must explicitly code the transaction that wraps the update process as well as explicitly call *CommitUpdates* to commit updates from the cache.

**Important** To apply updates from a stored procedure or an SQL query that does not return a live result set, you must use *TUpdateSQL* to specify how to perform updates. For updates to joins (queries involving two or more tables), you must provide one *TUpdateSQL* object for each table involved, and you must use the *OnUpdateRecord* event handler to invoke these objects to perform the updates. See "Using update objects to update a dataset" on page 26-40 for details.

## Applying cached updates using a database

To apply cached updates to one or more datasets in the context of a database connection, call the database component's *ApplyUpdates* method. The following code applies updates to the *CustomersQuery* dataset in response to a button click event:

```
procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
  // for local databases such as Paradox, dBASE, and FoxPro
  // set TransIsolation to DirtyRead
  if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
    Database1.TransIsolation := tiDirtyRead;
  Database1.ApplyUpdates([CustomersQuery]);
end;
```

The above sequence writes cached updates to the database in the context of an automatically-generated transaction. If successful, it commits the transaction and then commits the cached updates. If unsuccessful, it rolls back the transaction and leaves the update cache unchanged. In this latter case, you should handle cached update errors through a dataset's *OnUpdateError* event. For more information about handling update errors, see "Handling cached update errors" on page 26-38.

The main advantage to calling a database component's *ApplyUpdates* method is that you can update any number of dataset components that are associated with the database. The parameter for the *ApplyUpdates* method for a database is an array of *TDBDataSet*. For example, the following code applies updates for two queries:

```
if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
  Database1.TransIsolation := tiDirtyRead;
Database1.ApplyUpdates([CustomerQuery, OrdersQuery]);
```

## Applying cached updates with dataset component methods

You can apply updates for individual BDE-enabled datasets directly using the dataset's *ApplyUpdates* and *CommitUpdates* methods. Each of these methods encapsulate one phase of the update process:

1. *ApplyUpdates* writes cached changes to a database (phase 1).
2. *CommitUpdates* clears the internal cache when the database write is successful (phase 2).

The following code illustrates how you apply updates within a transaction for the *CustomerQuery* dataset:

```
procedure TForm1.ApplyButtonClick(Sender: TObject)
begin
  Database1.StartTransaction;
  try
    if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
      Database1.TransIsolation := tiDirtyRead;
    CustomerQuery.ApplyUpdates;                 { try to write the updates to the database }
    Database1.Commit;                                      { on success, commit the changes }
  except
    Database1.Rollback;                                  { on failure, undo any changes }
    raise;                    { raise the exception again to prevent a call to CommitUpdates }
  end;
  CustomerQuery.CommitUpdates;                       { on success, clear the internal cache }
end;
```

If an exception is raised during the *ApplyUpdates* call, the database transaction is rolled back. Rolling back the transaction ensures that the underlying database table is not changed. The *raise* statement inside the try...except block re*raises* the exception, thereby preventing the call to *CommitUpdates*. Because *CommitUpdates* is not called, the internal cache of updates is not cleared so that you can handle error conditions and possibly retry the update.

## Creating an OnUpdateRecord event handler

When a BDE-enabled dataset applies its cached updates, it iterates through the changes recorded in its cache, attempting to apply them to the corresponding records in the base table. As the update for each changed, deleted, or newly inserted record is about to be applied, the dataset component's *OnUpdateRecord* event fires.

Providing a handler for the *OnUpdateRecord* event allows you to perform actions just before the current record's update is actually applied. Such actions can include special data validation, updating other tables, special parameter substitution, or executing multiple update objects. A handler for the *OnUpdateRecord* event affords you greater control over the update process.

Here is the skeleton code for an *OnUpdateRecord* event handler:

```
procedure TForm1.DataSetUpdateRecord(DataSet: TDataSet;
UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { perform updates here... }
end;
```

The *DataSet* parameter specifies the cached dataset with updates.

The *UpdateKind* parameter indicates the type of update that needs to be performed for the current record. Values for *UpdateKind* are *ukModify*, *ukInsert*, and *ukDelete*. If you are using an update object, you need to pass this parameter to the update object when applying the update. You may also need to inspect this parameter if your handler performs any special processing based on the kind of update.

The *UpdateAction* parameter indicates whether you applied the update. Values for *UpdateAction* are *uaFail* (the default), *uaAbort*, *uaSkip*, *uaRetry*, *uaApplied*. If your event handler successfully applies the update, change this parameter to *uaApplied* before exiting. If you decide not to update the current record, change the value to *uaSkip* to preserve unapplied changes in the cache. If you do not change the value for *UpdateAction*, the entire update operation for the dataset is aborted and an exception is raised. You can suppress the error message (raising a silent exception) by changing *UpdateAction* to *uaAbort*.

In addition to these parameters, you will typically want to make use of the *OldValue* and *NewValue* properties for the field component associated with the current record. *OldValue* gives the original field value that was fetched from the database. It can be useful in locating the database record to update. *NewValue* is the edited value in the update you are trying to apply.

**Important** An *OnUpdateRecord* event handler, like an *OnUpdateError* or *OnCalcFields* event handler, should never call any methods that change the current record in a dataset.

The following example illustrates how to use these parameters and properties. It uses a *TTable* component named *UpdateTable* to apply updates. In practice, it is easier to use an update object, but using a table illustrates the possibilities more clearly.

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  if UpdateKind = ukInsert then
    UpdateTable.AppendRecord([DataSet.Fields[0].NewValue, DataSet.Fields[1].NewValue])
  else
    if UpdateTable.Locate('KeyField', VarToStr(DataSet.Fields[1].OldValue), []) then
      case UpdateKind of
        ukModify:
          begin
            UpdateTable.Edit;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            UpdateTable.Post;
          end;
        ukInsert:
          begin
            UpdateTable.Insert;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            UpdateTable.Post;
          end;
        ukDelete: UpdateTable.Delete;
      end;
        UpdateAction := uaApplied;
end;
```

## Handling cached update errors

The Borland Database Engine (BDE) specifically checks for user update conflicts and other conditions when attempting to apply updates, and reports any errors. The dataset component's *OnUpdateError* event enables you to catch and respond to errors. You should create a handler for this event if you use cached updates. If you do not, and an error occurs, the entire update operation fails.

Here is the skeleton code for an *OnUpdateError* event handler:

```
procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E: EDatabaseError;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { ... perform update error handling here ... }
end;
```

*DataSet* references the dataset to which updates are applied. You can use this dataset to access new and old values during error handling. The original values for fields in each record are stored in a read-only *TField* property called *OldValue*. Changed values are stored in the analogous *TField* property *NewValue*. These values provide the only way to inspect and change update values in the event handler.

**Warning** Do not call any dataset methods that change the current record (such as *Next* and *Prior*). Doing so causes the event handler to enter an endless loop.

The *E* parameter is usually of type *EDBEngineError*. From this exception type, you can extract an error message that you can display to users in your error handler. For example, the following code could be used to display the error message in the caption of a dialog box:

```
ErrorLabel.Caption := E.Message;
```

This parameter is also useful for determining the actual cause of the update error. You can extract specific error codes from *EDBEngineError*, and take appropriate action based on it.

The *UpdateKind* parameter describes the type of update that generated the error. Unless your error handler takes special actions based on the type of update being carried out, your code probably will not make use of this parameter.

The following table lists possible values for *UpdateKind*:

**Table 26.7** UpdateKind values

| Value | Meaning |
|---|---|
| *ukModify* | Editing an existing record caused an error. |
| *ukInsert* | Inserting a new record caused an error. |
| *ukDelete* | Deleting an existing record caused an error. |

*UpdateAction* tells the BDE how to proceed with the update process when your event handler exits. When your update error handler is first called, the value for this parameter is always set to *uaFail*. Based on the error condition for the record that caused the error and what you do to correct it, you typically set *UpdateAction* to a different value before exiting the handler:

- If your error handler can correct the error condition that caused the handler to be invoked, set *UpdateAction* to the appropriate action to take on exit. For error conditions you correct, set *UpdateAction* to *uaRetry* to apply the update for the record again.
- When set to *uaSkip*, the update for the row that caused the error is skipped, and the update for the record remains in the cache after all other updates are completed.

- Both *uaFail* and *uaAbort* cause the entire update operation to end. *uaFail* raises an exception and displays an error message. *uaAbort* raises a silent exception (does not display an error message).

The following code shows an *OnUpdateError* event handler that checks to see if the update error is related to a key violation, and if it is, it sets the *UpdateAction* parameter to *uaSkip*:

```
{ Add 'Bde' to your uses clause for this example }
if (E is EDBEngineError) then
  with EDBEngineError(E) do begin
    if Errors[ErrorCount - 1].ErrorCode = DBIERR_KEYVIOL then
      UpdateAction := uaSkip                         { key violation, just skip this record }
    else
      UpdateAction := uaAbort;                  { don't know what's wrong, abort the update }
  end;
```

**Note** If an error occurs during the application of cached updates, an exception is *raised* and an error message displayed. Unless the *ApplyUpdates* is called from within a try...except construct, an error message to the user displayed from inside your *OnUpdateError* event handler may cause your application to display the same error message twice. To prevent error message duplication, set *UpdateAction* to *uaAbort* to turn off the system-generated error message display.

## Using update objects to update a dataset

When the BDE-enabled dataset represents a stored procedure or a query that is not “live”, it is not possible to apply updates directly from the dataset. Such datasets may also cause a problem when you use a client dataset to cache updates. Whether you are using the BDE or a client dataset to cache updates, you can handle these problem datasets by using an update object:

1. If you are using a client dataset, use an external provider component with *TClientDataSet* rather than *TBDEClientDataSet*. This is so you can set the *UpdateObject* property of the BDE-enabled source dataset (step 3).
2. Add a *TUpdateSQL* component to the same data module as the BDE-enabled dataset.
3. Set the BDE-enabled dataset component’s *UpdateObject* property to the *TUpdateSQL* component in the data module.
4. Specify the SQL statements needed to perform updates using the update object’s *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties. You can use the Update SQL editor to help you compose these statements.
5. Close the dataset.
6. Set the dataset component’s *CachedUpdates* property to *True* or link the dataset to the client dataset using a dataset provider.
7. Reopen the dataset.

**Note** Sometimes, you need to use multiple update objects. For example, when updating a multi-table join or a stored procedure that represents data from multiple datasets, you must provide one *TUpdateSQL* object for each table you want to update. When using multiple update objects, you can't simply associate the update object with the dataset by setting the *UpdateObject* property. Instead, you must manually call the update object from an *OnUpdateRecord* event handler (when using the BDE to cache updates) or a *BeforeUpdateRecord* event handler (when using a client dataset).

The update object actually encapsulates three *TQuery* components. Each of these query components perform a single update task. One query component provides an SQL UPDATE statement for modifying existing records; a second query component provides an INSERT statement to add new records to a table; and a third component provides a DELETE statement to remove records from a table.

When you place an update component in a data module, you do not see the query components it encapsulates. They are created by the update component at runtime based on three update properties for which you supply SQL statements:

- *ModifySQL* specifies the UPDATE statement.
- *InsertSQL* specifies the INSERT statement.
- *DeleteSQL* specifies the DELETE statement.

At runtime, when the update component is used to apply updates, it:

1 Selects an SQL statement to execute based on whether the current record is modified, inserted, or deleted.

2 Provides parameter values to the SQL statement.

3 Prepares and executes the SQL statement to perform the specified update.

### Creating SQL statements for update components

To update a record in an associated dataset, an update object uses one of three SQL statements. Each update object can only update a single table, so the object's update statements must each reference the same base table.

The three SQL statements delete, insert, and modify records cached for update. You must provide these statements as update object's *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties. You can provide these values at design time or at runtime. For example, the following code specifies a value for the *DeleteSQL* property at runtime:

```
with UpdateSQL1.DeleteSQL do begin
  Clear;
  Add('DELETE FROM Inventory I');
  Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;
```

At design time, you can use the Update SQL editor to help you compose the SQL statements that apply updates.

Update objects provide automatic parameter binding for parameters that reference the dataset's original and updated field values. Typically, therefore, you insert parameters with specially formatted names when you compose the SQL statements. For information on using these parameters, see "Understanding parameter substitution in update SQL statements" on page 26-43.

### Using the Update SQL editor

To create the SQL statements for an update component,

1 Using the Object Inspector, select the name of the update object from the drop-down list for the dataset's *UpdateObject* property. This step ensures that the Update SQL editor you invoke in the next step can determine suitable default values to use for SQL generation options.

2 Right-click the update object and select UpdateSQL Editor from the context menu. This displays the Update SQL editor. The editor creates SQL statements for the update object's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties based on the underlying data set and on the values you supply to it.

The Update SQL editor has two pages. The Options page is visible when you first invoke the editor. Use the Table Name combo box to select the table to update. When you specify a table name, the Key Fields and Update Fields list boxes are populated with available columns.

The Update Fields list box indicates which columns should be updated. When you first specify a table, all columns in the Update Fields list box are selected for inclusion. You can multi-select fields as desired.

The Key Fields list box is used to specify the columns to use as keys during the update. For Paradox, dBASE, and FoxPro the columns you specify here must correspond to an existing index, but this is not a requirement for remote SQL databases. Instead of setting Key Fields you can click the Primary Keys button to choose key fields for the update based on the table's primary index. Click Dataset Defaults to return the selection lists to the original state: all fields selected as keys and all selected for update.

Check the Quote Field Names check box if your server requires quotation marks around field names.

After you specify a table, select key columns, and select update columns, click Generate SQL to generate the preliminary SQL statements to associate with the update component's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties. In most cases you will want or need to fine tune the automatically generated SQL statements.

To view and modify the generated SQL statements, select the SQL page. If you have generated SQL statements, then when you select this page, the statement for the *ModifySQL* property is already displayed in the SQL Text memo box. You can edit the statement in the box as desired.

**Important** Keep in mind that generated SQL statements are starting points for creating update statements. You may need to modify these statements to make them execute correctly. For example, when working with data that contains NULL values, you need to modify the WHERE clause to read

```
WHERE field IS NULL
```

rather then using the generated field variable. Test each of the statements directly yourself before accepting them.

Use the Statement Type radio buttons to switch among generated SQL statements and edit them as desired.

To accept the statements and associate them with the update component's SQL properties, click OK.

### Understanding parameter substitution in update SQL statements

Update SQL statements use a special form of parameter substitution that enables you to substitute old or new field values in record updates. When the Update SQL editor generates its statements, it determines which field values to use. When you write the update SQL, you specify the field values to use.

When the parameter name matches a column name in the table, the new value in the field in the cached update for the record is automatically used as the value for the parameter. When the parameter name matches a column name prefixed by the string "OLD_", then the old value for the field will be used. For example, in the update SQL statement below, the parameter :LastName is automatically filled with the new field value in the cached update for the inserted record.

```
INSERT INTO Names
(LastName, FirstName, Address, City, State, Zip)
VALUES (:LastName, :FirstName, :Address, :City, :State, :Zip)
```

New field values are typically used in the *InsertSQL* and *ModifySQL* statements. In an update for a modified record, the new field value from the update cache is used by the UPDATE statement to replace the old field value in the base table updated.

In the case of a deleted record, there are no new values, so the *DeleteSQL* property uses the ":OLD_FieldName" syntax. Old field values are also normally used in the WHERE clause of the SQL statement for a modified or deletion update to determine which record to update or delete.

In the WHERE clause of an UPDATE or DELETE update SQL statement, supply at least the minimal number of parameters to uniquely identify the record in the base table that is updated with the cached data. For instance, in a list of customers, using just a customer's last name may not be sufficient to uniquely identify the correct record in the base table; there may be a number of records with "Smith" as the last name. But by using parameters for last name, first name, and phone number could be a distinctive enough combination. Even better would be a unique field value like a customer number.

**Note** If you create SQL statements that contain parameters that do not refer the edited or original field values, the update object does not know how to bind their values. You can, however, do this manually, using the update object's *Query* property. See "Using an update component's Query property" on page 26-48 for details.

### Composing update SQL statements

At design time, you can use the Update SQL editor to write the SQL statements for the *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties. If you do not use the Update SQL editor, or if you want to modify the generated statements, you should keep in mind the following guidelines when writing statements to delete, insert, and modify records in the base table.

The *DeleteSQL* property should contain only an SQL statement with the DELETE command. The base table to be updated must be named in the FROM clause. So that the SQL statement only deletes the record in the base table that corresponds to the record deleted in the update cache, use a WHERE clause. In the WHERE clause, use a parameter for one or more fields to uniquely identify the record in the base table that corresponds to the cached update record. If the parameters are named the same as the field and prefixed with "OLD_", the parameters are automatically given the values from the corresponding field from the cached update record. If the parameter are named in any other manner, you must supply the parameter values.

```
DELETE FROM Inventory I
WHERE (I.ItemNo = :OLD_ItemNo)
```

Some table types might not be able to find the record in the base table when fields used to identify the record contain NULL values. In these cases, the delete update fails for those records. To accommodate this, add a condition for those fields that might contain NULLs using the IS NULL predicate (in addition to a condition for a non-NULL value). For example, when a FirstName field may contain a NULL value:

```
DELETE FROM Names
WHERE (LastName = :OLD_LastName) AND
  ((FirstName = :OLD_FirstName) OR (FirstName IS NULL))
```

The *InsertSQL* statement should contain only an SQL statement with the INSERT command. The base table to be updated must be named in the INTO clause. In the VALUES clause, supply a comma-separated list of parameters. If the parameters are named the same as the field, the parameters are automatically given the value from the cached update record. If the parameter are named in any other manner, you must supply the parameter values. The list of parameters supplies the values for fields in the newly inserted record. There must be as many value parameters as there are fields listed in the statement.

```
INSERT INTO Inventory
(ItemNo, Amount)
VALUES (:ItemNo, 0)
```

The *ModifySQL* statement should contain only an SQL statement with the UPDATE command. The base table to be updated must be named in the FROM clause. Include one or more value assignments in the SET clause. If values in the SET clause assignments are parameters named the same as fields, the parameters are automatically given values from the fields of the same name in the updated record in the cache. You can assign additional field values using other parameters, as long as the parameters are not named the same as any fields and you manually supply the values. As with the *DeleteSQL* statement, supply a WHERE clause to uniquely identify the record in the base table to be updated using parameters named the same as the fields and prefixed with "OLD_". In the update statement below, the parameter :ItemNo is automatically given a value and :Price is not.

```
UPDATE Inventory I
SET I.ItemNo = :ItemNo, Amount = :Price
WHERE (I.ItemNo = :OLD_ItemNo)
```

Considering the above update SQL, take an example case where the application end-user modifies an existing record. The original value for the ItemNo field is 999. In a grid connected to the cached dataset, the end-user changes the ItemNo field value to 123 and Amount to 20. When the ApplyUpdates method is invoked, this SQL statement affects all records in the base table where the ItemNo field is 999, using the old field value in the parameter :OLD_ItemNo. In those records, it changes the ItemNo field value to 123 (using the parameter :ItemNo, the value coming from the grid) and Amount to 20.

## Using multiple update objects

When more than one base table referenced in the update dataset needs to be updated, you need to use multiple update objects: one for each base table updated. Because the dataset component's *UpdateObject* only allows one update object to be associated with the dataset, you must associate each update object with a dataset by setting its *DataSet* property to the name of the dataset.

**Tip** When using multiple update objects, you can use *TBDEClientDataSet* instead of *TClientDataSet* with an external provider. This is because you do not need to set the source dataset's *UpdateObject* property.

The *DataSet* property for update objects is not available at design time in the Object Inspector. You can only set this property at runtime.

```
UpdateSQL1.DataSet := Query1;
```

The update object uses this dataset to obtain original and updated field values for parameter substitution and, if it is a BDE-enabled dataset, to identify the session and database to use when applying the updates. So that parameter substitution will work correctly, the update object's *DataSet* property must be the dataset that contains the updated field values. When using the BDE-enabled dataset to cache updates, this is the BDE-enabled dataset itself. When using a client dataset, this is a client dataset that is provided as a parameter to the *BeforeUpdateRecord* event handler.

When the update object has not been assigned to the dataset's *UpdateObject* property, its SQL statements are not automatically executed when you call *ApplyUpdates*. To update records, you must manually call the update object from an *OnUpdateRecord* event handler (when using the BDE to cache updates) or a *BeforeUpdateRecord* event handler (when using a client dataset). In the event handler, the minimum actions you need to take are

- If you are using a client dataset to cache updates, you must be sure that the updates object's *DatabaseName* and *SessionName* properties are set to the *DatabaseName* and *SessionName* properties of the source dataset.
- The event handler must call the update object's *ExecSQL* or *Apply* method. This invokes the update object for each record that requires updating. For more information about executing update statements, see "Executing the SQL statements" below.
- Set the event handler's *UpdateAction* parameter to *uaApplied* (*OnUpdateRecord*) or the *Applied* parameter to *True* (*BeforeUpdateRecord*).

You may optionally perform data validation, data modification, or other operations that depend on each record's update.

**Warning** If you call an update object's *ExecSQL* or *Apply* method in an *OnUpdateRecord* event handler, be sure that you do not set the dataset's *UpdateObject* property to that update object. Otherwise, this will result in a second attempt to apply each record's update.

### Executing the SQL statements

When you use multiple update objects, you do not associate the update objects with a dataset by setting its *UpdateObject* property. As a result, the appropriate statements are not automatically executed when you apply updates. Instead, you must explicitly invoke the update object in code.

There are two ways to invoke the update object. Which way you choose depends on whether the SQL statement uses parameters to represent field values:

- If the SQL statement to execute uses parameters, call the *Apply* method.
- If the SQL statement to execute does not use parameters, it is more efficient to call the *ExecSQL* method.

**Note** If the SQL statement uses parameters other than the built-in types (for the original and updated field values), you must manually supply parameter values instead of relying on the parameter substitution provided by the *Apply* method. See "Using an update component's Query property" on page 26-48 for information on manually providing parameter values.

For information about the default parameter substitution for parameters in an update object's SQL statements, see "Understanding parameter substitution in update SQL statements" on page 26-43.

#### Calling the Apply method

The *Apply* method for an update component manually applies updates for the current record. There are two steps involved in this process:

1. Initial and edited field values for the record are bound to parameters in the appropriate SQL statement.
2. The SQL statement is executed.

Call the *Apply* method to apply the update for the current record in the update cache. The *Apply* method is most often called from within a handler for the dataset's *OnUpdateRecord* event or from a provider's *BeforeUpdateRecord* event handler.

**Warning** If you use the dataset's *UpdateObject* property to associate dataset and update object, *Apply* is called automatically. In that case, do not call *Apply* in an *OnUpdateRecord* event handler as this will result in a second attempt to apply the current record's update.

*OnUpdateRecord* event handlers indicate the type of update that needs to be applied with an *UpdateKind* parameter of type *TUpdateKind*. You must pass this parameter to the *Apply* method to indicate which update SQL statement to use. The following code illustrates this using a *BeforeUpdateRecord* event handler:

```
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
          DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  with UpdateSQL1 do
  begin
    DataSet := DeltaDS;
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    Apply(UpdateKind);
    Applied := True;
  end;
end;
```

## Calling the ExecSQL method

The *ExecSQL* method for an update component manually applies updates for the current record. Unlike the *Apply* method, *ExecSQL* does not bind parameters in the SQL statement before executing it. The *ExecSQL* method is most often called from within a handler for the *OnUpdateRecord* event (when using the BDE) or the *BeforeUpdateRecord* event (when using a client dataset).

Because *ExecSQL* does not bind parameter values, it is used primarily when the update object's SQL statements do not include parameters. You can use *Apply* instead, even when there are no parameters, but *ExecSQL* is more efficient because it does not check for parameters.

If the SQL statements include parameters, you can still call *ExecSQL*, but only after explicitly binding parameters. If you are using the BDE to cache updates, you can explicitly bind parameters by setting the update object's *DataSet* property and then calling its *SetParams* method. When using a client dataset to cache updates, you must supply parameters to the underlying query object maintained by *TUpdateSQL*. For information on how to do this, see "Using an update component's Query property" on page 26-48.

**Warning** If you use the dataset's *UpdateObject* property to associate dataset and update object, *ExecSQL* is called automatically. In that case, do not call *ExecSQL* in an *OnUpdateRecord* or *BeforeUpdateRecord* event handler as this will result in a second attempt to apply the current record's update.

*OnUpdateRecord* and *BeforeUpdateRecord* event handlers indicate the type of update that needs to be applied with an *UpdateKind* parameter of type *TUpdateKind*. You must pass this parameter to the *ExecSQL* method to indicate which update SQL statement to use. The following code illustrates this using a *BeforeUpdateRecord* event handler:

```
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
          DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
```

```
begin
  with UpdateSQL1 do
  begin
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    ExecSQL(UpdateKind);
    Applied := True;
  end;
end;
```

If an exception is raised during the execution of the update program, execution continues in the *OnUpdateError* event, if it is defined.

### Using an update component's Query property

The *Query* property of an update component provides access to the query components that implement its *DeleteSQL*, *InsertSQL*, and *ModifySQL* statements. In most applications, there is no need to access these query components directly: you can use the *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties to specify the statements these queries execute, and execute them by calling the update object's *Apply* or *ExecSQL* method. There are times, however, when you may need to directly manipulate the query component. In particular, the *Query* property is useful when you want to supply your own values for parameters in the SQL statements rather than relying on the update object's automatic parameter binding to old and new field values.

**Note** The *Query* property is only accessible at runtime.

The *Query* property is indexed on a *TUpdateKind* value:

- Using an index of *ukModify* accesses the query that updates existing records.
- Using an index of *ukInsert* accesses the query that inserts new records.
- Using an index of *ukDelete* accesses the query that deletes records.

The following shows how to use the *Query* property to supply parameter values that can't be bound automatically:

```
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
          DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  UpdateSQL1.DataSet := DeltaDS; { required for the automatic parameter substitution }
  with UpdateSQL1.Query[UpdateKind] do
  begin
    { Make sure the query has the correct DatabaseName and SessionName }
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    ParamByName('TimeOfUpdate').Value = Now;
  end;
  UpdateSQL1.Apply(UpdateKind); { now perform automatic substitutions and execute }
  Applied := True;
end;
```

## Using TBatchMove

*TBatchMove* encapsulates Borland Database Engine (BDE) features that let you to duplicate a dataset, append records from one dataset to another, update records in one dataset with records from another dataset, and delete records from one dataset that match records in another dataset. *TBatchMove* is most often used to:

- Download data from a server to a local data source for analysis or other operations.
- Move a desktop database into tables on a remote server as part of an upsizing operation.

A batch move component can create tables on the destination that correspond to the source tables, automatically mapping the column names and data types as appropriate.

### Creating a batch move component

To create a batch move component:

1. Place a table or query component for the dataset from which you want to import records (called the *Source* dataset) on a form or in a data module.
2. Place the dataset to which to move records (called the *Destination* dataset) on the form or data module.
3. Place a *TBatchMove* component from the BDE page of the Component palette in the data module or form, and set its *Name* property to a unique value appropriate to your application.
4. Set the *Source* property of the batch move component to the name of the table from which to copy, append, or update records. You can select tables from the drop-down list of available dataset components.
5. Set the *Destination* property to the dataset to create, append to, or update. You can select a destination table from the drop-down list of available dataset components.
   - If you are appending, updating, or deleting, *Destination* must represent an existing database table.
   - If you are copying a table and *Destination* represents an existing table, executing the batch move overwrites all of the current data in the destination table.
   - If you are creating an entirely new table by copying an existing table, the resulting table has the name specified in the *Name* property of the table component to which you are copying. The resulting table type will be of a structure appropriate to the server specified by the *DatabaseName* property.
6. Set the *Mode* property to indicate the type of operation to perform. Valid operations are *batAppend* (the default), *batUpdate*, *batAppendUpdate*, *batCopy*, and *batDelete*. For information about these modes, see “Specifying a batch move mode” on page 26-50.

7 Optionally set the *Transliterate* property. If *Transliterate* is *True* (the default), character data is translated from the *Source* dataset's character set to the *Destination* dataset's character set as necessary.

8 Optionally set column mappings using the *Mappings* property. You need not set this property if you want batch move to match columns based on their position in the source and destination tables. For more information about mapping columns, see "Mapping data types" on page 26-51.

9 Optionally specify the *ChangedTableName*, *KeyViolTableName*, and *ProblemTableName* properties. Batch move stores problem records it encounters during the batch operation in the table specified by *ProblemTableName*. If you are updating a Paradox table through a batch move, key violations can be reported in the table you specify in *KeyViolTableName*. *ChangedTableName* lists all records that changed in the destination table as a result of the batch move operation. If you do not specify these properties, these error tables are not created or used. For more information about handling batch move errors, see "Handling batch move errors" on page 26-52.

## Specifying a batch move mode

The *Mode* property specifies the operation a batch move component performs:

**Table 26.8** Batch move modes

| Property | Purpose |
|---|---|
| batAppend | Append records to the destination table. |
| batUpdate | Update records in the destination table with matching records from the source table. Updating is based on the current index of the destination table. |
| batAppendUpdate | If a matching record exists in the destination table, update it. Otherwise, append records to the destination table. |
| batCopy | Create the destination table based on the structure of the source table. If the destination table already exists, it is dropped and recreated. |
| batDelete | Delete records in the destination table that match records in the source table. |

### Appending records

To append data, the destination dataset must represent an existing table. During the append operation, the BDE converts data to appropriate data types and sizes for the destination dataset if necessary. If a conversion is not possible, an exception is thrown and the data is not appended.

### Updating records

To update data, the destination dataset must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are overwritten with the source data. During the update operation, the BDE converts data to appropriate data types and sizes for the destination dataset if necessary.

### Appending and updating records

To append and update data the destination dataset must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are overwritten with the source data. Otherwise, data from the source dataset is appended to the destination dataset. During append and update operations, the BDE converts data to appropriate data types and sizes for the destination dataset, if necessary.

### Copying datasets

To copy a source dataset, the destination dataset should not represent an exist table. If it does, the batch move operation overwrites the existing table with a copy of the source dataset.

If the source and destination datasets are maintained by different types of database engines, for example, Paradox and InterBase, the BDE creates a destination dataset with a structure as close as possible to that of the source dataset and automatically performs data type and size conversions as necessary.

**Note** *TBatchMove* does not copy metadata structures such as indexes, constraints, and stored procedures. You must recreate these metadata objects on your database server or through the SQL Explorer as appropriate.

### Deleting records

To delete data in the destination dataset, it must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are deleted in the destination table.

## Mapping data types

In *batAppend* mode, a batch move component creates the destination table based on the column data types of the source table. Columns and types are matched based on their position in the source and destination tables. That is, the first column in the source is matched with the first column in the destination, and so on.

To override the default column mappings, use the *Mappings* property. *Mappings* is a list of column mappings (one per line). This listing can take one of two forms. To map a column in the source table to a column of the same name in the destination table, you can use a simple listing that specifies the column name to match. For example, the following mapping specifies that a column named *ColName* in the source table should be mapped to a column of the same name in the destination table:

```
ColName
```

To map a column named *SourceColName* in the source table to a column named *DestColName* in the destination table, the syntax is as follows:

```
DestColName = SourceColName
```

If source and destination column data types are not the same, a batch move operation attempts a “best fit”. It trims character data types, if necessary, and attempts to perform a limited amount of conversion, if possible. For example, mapping a CHAR(10) column to a CHAR(5) column will result in trimming the last five characters from the source column.

As an example of conversion, if a source column of character data type is mapped to a destination of integer type, the batch move operation converts a character value of ‘5’ to the corresponding integer value. Values that cannot be converted generate errors. For more information about errors, see “Handling batch move errors” on page 26-52.

When moving data between different table types, a batch move component translates data types as appropriate based on the dataset’s server types. See the BDE online help file for the latest tables of mappings among server types.

**Note** To batch move data to an SQL server database, you must have that database server and a version of Delphi with the appropriate SQL Link installed, or you can use ODBC if you have the proper third party ODBC drivers installed.

## Executing a batch move

Use the *Execute* method to execute a previously prepared batch operation at runtime. For example, if *BatchMoveAdd* is the name of a batch move component, the following statement executes it:

```
BatchMoveAdd.Execute;
```

You can also execute a batch move at design time by right clicking the mouse on a batch move component and choosing Execute from the context menu.

The *MovedCount* property keeps track of the number of records that are moved when a batch move executes.

The *RecordCount* property specifies the maximum number of records to move. If *RecordCount* is zero, all records are moved, beginning with the first record in the source dataset. If *RecordCount* is a positive number, a maximum of *RecordCount* records are moved, beginning with the current record in the source dataset. If *RecordCount* is greater than the number of records between the current record in the source dataset and its last record, the batch move terminates when the end of the source dataset is reached. You can examine *MoveCount* to determine how many records were actually transferred.

## Handling batch move errors

There are two types of errors that can occur in a batch move operation: data type conversion errors and integrity violations. *TBatchMove* has a number of properties that report on and control error handling.

The *AbortOnProblem* property specifies whether to abort the operation when a data type conversion error occurs. If *AbortOnProblem* is *True*, the batch move operation is canceled when an error occurs. If *False*, the operation continues. You can examine the table you specify in the *ProblemTableName* to determine which records caused problems.

The *AbortOnKeyViol* property indicates whether to abort the operation when a Paradox key violation occurs.

The *ProblemCount* property indicates the number of records that could not be handled in the destination table without a loss of data. If *AbortOnProblem* is *True*, this number is one, since the operation is aborted when an error occurs.

The following properties enable a batch move component to create additional tables that document the batch move operation:

- *ChangedTableName*, if specified, creates a local Paradox table containing all records in the destination table that changed as a result of an update or delete operation.
- *KeyViolTableName,* if specified, creates a local Paradox table containing all records from the source table that caused a key violation when working with a Paradox table. If *AbortOnKeyViol* is *True*, this table will contain at most one entry since the operation is aborted on the first problem encountered.
- *ProblemTableName,* if specified, creates a local Paradox table containing all records that could not be posted in the destination table due to data type conversion errors. For example, the table could contain records from the source table whose data had to be trimmed to fit in the destination table. If *AbortOnProblem* is *True*, there is at most one record in this table since the operation is aborted on the first problem encountered.

**Note** If *ProblemTableName* is not specified, the data in the record is trimmed and placed in the destination table.

## The Data Dictionary

When you use the BDE to access your data, your application has access to the Data Dictionary. The Data Dictionary provides a customizable storage area, independent of your applications, where you can create extended field attribute sets that describe the content and appearance of data.

For example, if you frequently develop financial applications, you may create a number of specialized field attribute sets describing different display formats for currency. When you create datasets for your application at design time, rather than using the Object Inspector to set the currency fields in each dataset by hand, you can associate those fields with an extended field attribute set in the data dictionary. Using the data dictionary ensures a consistent data appearance within and across the applications you create.

In a client/server environment, the Data Dictionary can reside on a remote server for additional sharing of information.

To learn how to create extended field attribute sets from the Fields editor at design time, and how to associate them with fields throughout the datasets in your application, see "Creating attribute sets for field components" on page 25-13. To learn more about creating a data dictionary and extended field attributes with the SQL and Database Explorers, see their respective online help files.

A programming interface to the Data Dictionary is available in the drintf unit (located in the lib directory). This interface supplies the following methods:

**Table 26.9** Data Dictionary interface

| Routine | Use |
|---|---|
| DictionaryActive | Indicates if the data dictionary is active. |
| DictionaryDeactivate | Deactivates the data dictionary. |
| IsNullID | Indicates whether a given ID is a null ID |
| FindDatabaseID | Returns the ID for a database given its alias. |
| FindTableID | Returns the ID for a table in a specified database. |
| FindFieldID | Returns the ID for a field in a specified table. |
| FindAttrID | Returns the ID for a named attribute set. |
| GetAttrName | Returns the name an attribute set given its ID. |
| GetAttrNames | Executes a callback for each attribute set in the dictionary. |
| GetAttrID | Returns the ID of the attribute set for a specified field. |
| NewAttr | Creates a new attribute set from a field component. |
| UpdateAttr | Updates an attribute set to match the properties of a field. |
| CreateField | Creates a field component based on stored attributes. |
| UpdateField | Changes the properties of a field to match a specified attribute set. |
| AssociateAttr | Associates an attribute set with a given field ID. |
| UnassociateAttr | Removes an attribute set association for a field ID. |
| GetControlClass | Returns the control class for a specified attribute ID. |
| QualifyTableName | Returns a fully qualified table name (qualified by user name). |
| QualifyTableNameByName | Returns a fully qualified table name (qualified by user name). |
| HasConstraints | Indicates whether the dataset has constraints in the dictionary. |
| UpdateConstraints | Updates the imported constraints of a dataset. |
| UpdateDataset | Updates a dataset to the current settings and constraints in the dictionary. |

## Tools for working with the BDE

One advantage of using the BDE as a data access mechanism is the wealth of supporting utilities that ship with Delphi. These utilities include:

- **SQL Explorer** and **Database Explorer**: Delphi ships with one of these two applications, depending on which version you have purchased. Both Explorers enable you to
    - Examine existing database tables and structures. The SQL Explorer lets you examine and query remote SQL databases.
    - Populate tables with data
    - Create extended field attribute sets in the Data Dictionary or associate them with fields in your application.
    - Create and manage BDE aliases.

  SQL Explorer lets you do the following as well:
    - Create SQL objects such as stored procedures on remote database servers.
    - View the reconstructed text of SQL objects on remote database servers.
    - Run SQL scripts.
- **SQL Monitor**: SQL Monitor lets you watch all of the communication that passes between the remote database server and the BDE. You can filter the messages you want to watch, limiting them to only the categories of interest. SQL Monitor is most useful when debugging your application.
- **BDE Administration utility:** The BDE Administration utility lets you add new database drivers, configure the defaults for existing drivers, and create new BDE aliases.
- **Database Desktop**: If you are using Paradox or dBASE tables, Database Desktop lets you view and edit their data, create new tables, and restructure existing tables. Using Database Desktop affords you more control than using the methods of a *TTable* component (for example, it allows you to specify validity checks and language drivers). It provides the only mechanism for restructuring Paradox and dBASE tables other than making direct calls the BDE’s API.

Chapter

27

# Working with ADO components

The *dbGo* components provide data access through the ADO framework. ADO, (Microsoft ActiveX Data Objects) is a set of COM objects that access data through an OLE DB provider. The *dbGo* components encapsulate these ADO objects in the Delphi database architecture.

The ADO layer of an ADO-based application consists of Microsoft ADO 2.1, an OLE DB provider or ODBC driver for the data store access, client software for the specific database system used (in the case of SQL databases), a database back-end system accessible to the application (for SQL database systems), and a database. All of these must be accessible to the ADO-based application for it to be fully functional.

The ADO objects that figure most prominently are the Connection, Command, and Recordset objects. These ADO objects are wrapped by the *TADOConnection*, *TADOCommand*, and ADO dataset components. The ADO framework includes other "helper" objects, like the Field and Properties objects, but these are typically not used directly in dbGo applications and are not wrapped by dedicated components.

This chapter presents the *dbGo* components and discusses the unique features they add to the common Delphi database architecture. Before reading about the features peculiar to the *dbGo* components, you should familiarize yourself with the common features of database connection components and datasets described in Chapter 23, "Connecting to databases" and Chapter 24, "Understanding datasets."

## Overview of ADO components

The ADO page of the Component palette hosts the *dbGo* components. These components let you connect to an ADO data store, execute commands, and retrieve data from tables in databases using the ADO framework. They require ADO 2.1 (or higher) to be installed on the host computer. Additionally, client software for the target database system (such as Microsoft SQL Server) must be installed, as well as an OLE DB driver or ODBC driver specific to the particular database system.

Most *dbGo* components have direct counterparts in the components available for other data access mechanisms: a database connection component (*TADOConnection*) and various types of datasets. In addition, *dbGo* includes *TADOCommand*, a simple component that is not a dataset but which represents an SQL command to be executed on the ADO data store.

The following table lists the ADO components.

**Table 27.1** ADO components

| Component | Use |
|---|---|
| *TADOConnection* | A database connection component that establishes a connection with an ADO data store; multiple ADO dataset and command components can share this connection to execute commands, retrieve data, and operate on metadata. |
| *TADODataSet* | The primary dataset for retrieving and operating on data; *TADODataSet* can retrieve data from a single or multiple tables; can connect directly to a data store or use a *TADOConnection* component. |
| *TADOTable* | A table-type dataset for retrieving and operating on a recordset produced by a single database table; *TADOTable* can connect directly to a data store or use a *TADOConnection* component. |
| *TADOQuery* | A query-type dataset for retrieving and operating on a recordset produced by a valid SQL statement; *TADOQuery* can also execute data definition language (DDL) SQL statements. It can connect directly to a data store or use a *TADOConnection* component |
| *TADOStoredProc* | A stored procedure-type dataset for executing stored procedures; *TADOStoredProc* executes stored procedures that may or may not retrieve data. It can connect directly to a data store or use a *TADOConnection* component. |
| *TADOCommand* | A simple component for executing commands (SQL statements that do not return result sets); *TADOCommand* can be used with a supporting dataset component, or retrieve a dataset from a table; It can connect directly to a data store or use a *TADOConnection* component. |

## Connecting to ADO data stores

dbGo applications use Microsoft ActiveX Data Objects (ADO) 2.1 to interact with an OLE DB provider that connects to a data store and accesses its data. One of the items a data store can represent is a database. An ADO-based application requires that ADO 2.1 be installed on the client computer. ADO and OLE DB is supplied by Microsoft and installed with Windows.

An ADO provider represents one of a number of types of access, from native OLE DB drivers to ODBC drivers. These drivers must be installed on the client computer. OLE DB drivers for various database systems are supplied by the database vendor or by a third-party. If the application uses an SQL database, such as Microsoft SQL Server or Oracle, the client software for that database system must also be installed on the client computer. Client software is supplied by the database vendor and installed from the database systems CD (or disk).

To connect your application with the data store, use an ADO connection component (*TADOConnection*). Configure the ADO connection component to use one of the available ADO providers. Although *TADOConnection* is not strictly required, because ADO command and dataset components can establish connections directly using their *ConnectionString* property, you can use *TADOConnection* to share a single connection among several ADO components. This can reduce resource consumption, and allows you to create transactions that span multiple datasets.

Like other database connection components, *TADOConnection* provides support for

- Controlling connections
- Controlling server login
- Managing transactions
- Working with associated datasets
- Sending commands to the server
- Obtaining metadata

In addition to these features that are common to all database connection components, *TADOConnection* provides its own support for

- A wide range of options you can use to fine-tune the connection.
- The ability to list the command objects that use the connection.
- Additional events when performing common tasks.

### Connecting to a data store using TADOConnection

One or more ADO dataset and command components can share a single connection to a data store by using *TADOConnection*. To do so, associated dataset and command components with the connection component through their *Connection* properties. At design-time, select the desired connection component from the drop-down list for the *Connection* property in the Object Inspector. At runtime, assign the reference to the *Connection* property. For example, the following line associates a *TADODataSet* component with a *TADOConnection* component.

```
ADODataSet1.Connection := ADOConnection1;
```

The connection component represents an ADO connection object. Before you can use the connection object to establish a connection, you must identify the data store to which you want to connect. Typically, you provide information using the *ConnectionString* property. *ConnectionString* is a semicolon delimited string that lists one or more named connection parameters. These parameters identify the data store by specifying either the name of a file that contains the connection information or the name of an ADO provider and a reference identifying the data store. Use the following, predefined parameter names to supply this information:

**Table 27.2** Connection parameters

| Parameter | Description |
|---|---|
| *Provider* | The name of a local ADO provider to use for the connection. |
| *Data Source* | The name of the data store. |
| *File name* | The name of a file containing connection information. |
| *Remote Provider* | The name of an ADO provider that resides on a remote machine. |
| *Remote Server* | The name of the remote server when using a remote provider. |

Thus, a typical value of *ConnectionString* has the form

```
Provider=MSDASQL.1;Data Source=MQIS
```

**Note** The connection parameters in *ConnectionString* do not need to include the *Provider* or *Remote Provider* parameter if you specify an ADO provider using the *Provider* property. Similarly, you do not need to specify the *Data Source* parameter if you use the *DefaultDatabase* property.

In addition, to the parameters listed above, *ConnectionString* can include any connection parameters peculiar to the specific ADO provider you are using. These additional connection parameters can include user ID and password if you want to hardcode the login information.

At design-time, you can use the Connection String Editor to build a connection string by selecting connection elements (like the provider and server) from lists. Click the ellipsis button for the *ConnectionString* property in the Object Inspector to launch the Connection String Editor, which is an ActiveX property editor supplied by ADO.

Once you have specified the *ConnectionString* property (and, optionally, the *Provider* property), you can use the ADO connection component to connect to or disconnect from the ADO data store, although you may first want to use other properties to fine-tune the connection. When connecting to or disconnecting from the data store, *TADOConnection* lets you respond to a few additional events beyond those common to all database connection components. These additional events are described in "Events when establishing a connection" on page 27-8 and "Events when disconnecting" on page 27-8.

**Note** If you do not explicitly activate the connection by setting the connection component's *Connected* property to *True*, it automatically establishes the connection when the first dataset component is opened or the first time you use an ADO command component to execute a command.

### Accessing the connection object

Use the *ConnectionObject* property of *TADOConnection* to access the underlying ADO connection object. Using this reference it is possible to access properties and call methods of the underlying ADO Connection object.

Using the underlying ADO Connection object requires a good working knowledge of ADO objects in general and the ADO Connection object in particular. It is not recommended that you use the Connection object unless you are familiar with Connection object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO Connection objects.

## Fine-tuning a connection

One advantage of using *TADOConnection* for establishing the connection to a data store instead of simply supplying a connection string for your ADO command and dataset components, is that it provides a greater degree of control over the conditions and attributes of the connection.

### Forcing asynchronous connections

Use the *ConnectOptions* property to force the connection to be asynchronous. Asynchronous connections allow your application to continue processing without waiting for the connection to be completely opened.

By default, *ConnectionOptions* is set to *coConnectUnspecified* which allows the server to decide the best type of connection. To explicitly make the connection asynchronous, set *ConnectOptions* to *coAsyncConnect*.

The example routines below enable and disable asynchronous connections in the specified connection component:

```
procedure TForm1.AsyncConnectButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    ConnectOptions := coAsyncConnect;
    Open;
  end;
end;
procedure TForm1.ServerChoiceConnectButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    ConnectOptions := coConnectUnspecified;
    Open;
  end;
end;
```

### Controlling time-outs

You can control the amount of time that can elapse before attempted commands and connections are considered failed and are aborted using the *ConnectionTimeout* and *CommandTimeout* properties.

*ConnectionTimeout* specifies the amount of time, in seconds, before an attempt to connect to the data store times out. If the connection does not successfully compile prior to expiration of the time specified in *ConnectionTimeout*, the connection attempt is canceled:

```
with ADOConnection1 do begin
  ConnectionTimeout := 10 {seconds};
  Open;
end;
```

*CommandTimeout* specifies the amount of time, in seconds, before an attempted command times out. If a command initiated by a call to the *Execute* method does not successfully complete prior to expiration of the time specified in *CommandTimeout*, the command is canceled and ADO generates an exception:

```
with ADOConnection1 do begin
  CommandTimeout := 10 {seconds};
  Execute('DROP TABLE Employee1997', cmdText, []);
end;
```

### Indicating the types of operations the connection supports

ADO connections are established using a specific mode, similar to the mode you use when opening a file. The connection mode determines the permissions available to the connection, and hence the types of operations (such as reading and writing) that can be performed using that connection.

Use the *Mode* property to indicate the connection mode. The possible values are listed in Table 27.3:

**Table 27.3** ADO connection modes

| Connect Mode | Meaning |
|---|---|
| cmUnknown | Permissions are not yet set for the connection or cannot be determined. |
| cmRead | Read-only permissions are available to the connection. |
| cmWrite | Write-only permissions are available to the connection. |
| cmReadWrite | Read/write permissions are available to the connection. |
| cmShareDenyRead | Prevents others from opening connections with read permissions. |
| cmShareDenyWrite | Prevents others from opening connection with write permissions. |
| cmShareExclusive | Prevents others from opening connection. |
| cmShareDenyNone | Prevents others from opening connection with any permissions. |

The possible values for *Mode* correspond to the *ConnectModeEnum* values of the *Mode* property on the underlying ADO connection object. See the Microsoft Data Access SDK help for more information on these values.

### Specifying whether the connection automatically initiates transactions

Use the *Attributes* property to control the connection component's use of retaining commits and retaining aborts. When the connection component uses retaining commits, then every time your application commits a transaction, a new transaction is automatically started. When the connection component uses retaining aborts, then every time your application rolls back a transaction, a new transaction is automatically started.

*Attributes* is a set that can contain one, both, or neither of the constants *xaCommitRetaining* and *xaAbortRetaining*. When *Attributes* contains *xaCommitRetaining*, the connection uses retaining commits. When *Attributes* contains *xaAbortRetaining*, it uses retaining aborts.

Check whether either retaining commits or retaining aborts is enabled using the *in* operator. Enable retaining commits or aborts by adding the appropriate value to the attributes property; disable them by subtracting the value. The example routines below respectively enable and disable retaining commits in an ADO connection component.

```
procedure TForm1.RetainingCommitsOnButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if not (xaCommitRetaining in Attributes) then
      Attributes := (Attributes + [xaCommitRetaining])
    Open;
  end;
end;
procedure TForm1.RetainingCommitsOffButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if (xaCommitRetaining in Attributes) then
      Attributes := (Attributes - [xaCommitRetaining]);
    Open;
  end;
end;
```

## Accessing the connection's commands

Like other database connection components, you can access the datasets associated with the connection using the *DataSets* and *DataSetCount* properties. However, *dbGo* also includes *TADOCommand* objects, which are not datasets, but which maintain a similar relationship to the connection component.

You can use the *Commands* and *CommandCount* properties of *TADOConnection* to access the associated ADO command objects in the same way you use the *DataSets* and *DataSetCount* properties to access the associated datasets. Unlike *DataSets* and *DataSetCount*, which only list active datasets, *Commands* and *CommandCount* provide references to all *TADOCommand* components associated with the connection component.

*Commands* is a zero-based array of references to ADO command components. *CommandCount* provides a total count of all of the commands listed in *Commands*. You can use these properties together to iterate through all the commands that use a connection component, as illustrated in the following code*:*

```
var
  i: Integer
begin
  for i := 0 to (ADOConnection1.CommandCount - 1) do
    ADOConnection1.Commands[i].Execute;
end;
```

## ADO connection events

In addition to the usual events that occur for all database connection components, *TADOConnection* generates a number of additional events that occur during normal usage.

### Events when establishing a connection

In addition to the *BeforeConnect* and *AfterConnect* events that are common to all database connection components, *TADOConnection* also generates an *OnWillConnect* and *OnConnectComplete* event when establishing a connection. These events occur after the *BeforeConnect* event.

- *OnWillConnect* occurs before the ADO provider establishes a connection. It lets you make last minute changes to the connection string, provide a user name and password if you are handling your own login support, force an asynchronous connection, or even cancel the connection before it is opened.
- *OnConnectComplete* occurs after the connection is opened. Because *TADOConnection* can represent asynchronous connections, you should use *OnConnectComplete*, which occurs after the connection is opened or has failed due to an error condition, instead of the *AfterConnect* event, which occurs after the connection component instructs the ADO provider to open a connection, but not necessarily after the connection is opened.

### Events when disconnecting

In addition to the *BeforeDisconnect* and *AfterDisconnect* events common to all database connection components, *TADOConnection* also generates an *OnDisconnect* event after closing a connection. *OnDisconnect* occurs after the connection is closed but before any associated datasets are closed and before the *AfterDisconnect* event.

### Events when managing transactions

The ADO connection component provides a number of events for detecting when transaction-related processes have been completed. These events indicate when a transaction process initiated by a *BeginTrans*, *CommitTrans*, and *RollbackTrans* method has been successfully completed at the data store.

- The *OnBeginTransComplete* event occurs when the data store has successfully started a transaction after a call to the *BeginTrans* method.
- The *OnCommitTransComplete* event occurs after a transaction is successfully committed due to a call to *CommitTrans*.
- The *OnRollbackTransComplete* event occurs after a transaction is successfully aborted due to a call to *RollbackTrans*.

### Other events

ADO connection components introduce two additional events you can use to respond to notifications from the underlying ADO connection object:

- The *OnExecuteComplete* event occurs after the connection component executes a command on the data store (for example, after calling the *Execute* method). *OnExecuteComplete* indicates whether the execution was successful.
- The *OnInfoMessage* event occurs when the underlying connection object provides detailed information after an operation is completed. The *OnInfoMessage* event handler receives the interface to an ADO Error object that contains the detailed information and a status code indicating whether the operation was successful.

## Using ADO datasets

ADO dataset components encapsulate the ADO Recordset object. They inherit the common dataset capabilities described in Chapter 24, “Understanding datasets,” using ADO to provide the implementation. In order to use an ADO dataset, you must familiarize yourself with these common features.

In addition to the common dataset features, all ADO datasets add properties, events, and methods for

- Connecting to an ADO datastore.
- Accessing the underlying Recordset object.
- Filtering records based on bookmarks.
- Fetching records asynchronously.
- Performing batch updates (caching updates).
- Using files on disk to store data.

There are four ADO datasets:

- *TADOTable*, a table-type dataset that represents all of the rows and columns of a single database table. See "Using table type datasets" on page 24-25 for information on using *TADOTable* and other table-type datasets.
- *TADOQuery*, a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any. See "Using query-type datasets" on page 24-42 for information on using *TADOQuery* and other query-type datasets.
- *TADOStoredProc*, a stored procedure-type dataset that executes a stored procedure defined on a database server. See "Using stored procedure-type datasets" on page 24-50 for information on using *TADOStoredProc* and other stored procedure-type datasets.
- *TADODataSet*, a general-purpose dataset that includes the capabilities of the other three types. See "Using TADODataSet" on page 27-16 for a description of features unique to *TADODataSet*.

**Note** When using ADO to access database information, you do not need to use a dataset such as *TADOQuery* to represent SQL commands that do not return a cursor. Instead, you can use *TADOCommand*, a simple component that is not a dataset. For details on *TADOCommand*, see "Using Command objects" on page 27-18.

## Connecting an ADO dataset to a data store

ADO datasets can connect to an ADO data store either collectively or individually.

When connecting datasets collectively, set the *Connection* property of each dataset to a *TADOConnection* component. Each dataset then uses the ADO connection component's connection.

```
ADODataSet1.Connection  := ADOConnection1;
ADODataSet2.Connection  := ADOConnection1;
ƒ
```

Among the advantages of connecting datasets collectively are:

- The datasets share the connection object's attributes.
- Only one connection need be set up: that of the *TADOConnection*.
- The datasets can participate in transactions.

For more information on using *TADOConnection* see "Connecting to ADO data stores" on page 27-3.

When connecting datasets individually, set the *ConnectionString* property of each dataset. Each dataset that uses *ConnectionString* establishes its own connection to the data store, independent of any other dataset connection in the application.

The *ConnectionString* property of ADO datasets works the same way as the *ConnectionString* property of *TADOConnection*: it is a set of semicolon-delimited connection parameters such as the following:

```
ADODataSet1.ConnectionString := 'Provider=YourProvider;Password=SecretWord;' +
  'User ID=JaneDoe;SERVER=PURGATORY;UID=JaneDoe;PWD=SecretWord;' +
  'Initial Catalog=Employee';
```

At design time you can use the Connection String Editor to help you build the connection string. For more information about connection strings, see "Connecting to a data store using TADOConnection" on page 27-3.

## Working with record sets

The *Recordset* property provides direct access to the ADO recordset object underlying the dataset component. Using this object, it is possible to access properties and call methods of the recordset object from an application. Use of *Recordset* to directly access the underlying ADO recordset object requires a good working knowledge of ADO objects in general and the ADO recordset object in specific. Using the recordset object directly is not recommended unless you are familiar with recordset object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO recordset objects.

The *RecordsetState* property indicates the current state of the underlying recordset object. *RecordsetState* corresponds to the *State* property of the ADO recordset object. The value of *RecordsetState* is either *stOpen, stExecuting,* or *stFetching*. (*TObjectState,* the type of the *RecordsetState* property, defines other values, but only *stOpen, stExecuting,* and *stFetching* pertain to recordsets.) A value of *stOpen* indicates that the recordset is currently idle. A value of *stExecuting* indicates that it is executing a command. A value of *stFetching* indicates that it is fetching rows from the associated table (or tables).

Use *RecordsetState* values to perform actions dependent on the current state of the dataset. For example, a routine that updates data might check the *RecordsetState* property to see whether the dataset is active and not in the process of other activities such as connecting or fetching data.

## Filtering records based on bookmarks

ADO datasets support the common dataset feature of using bookmarks to mark and return to specific records. Also like other datasets, ADO datasets let you use filters to limit the available records in the dataset. ADO datasets provide an additional feature that combines these two common dataset features: the ability to filter on a set of records identified by bookmarks.

To filter on a set of bookmarks,

1. Use the *Bookmark* method to mark the records you want to include in the filtered dataset.
2. Call the *FilterOnBookmarks* method to filter the dataset so that only the bookmarked records appear.

This process is illustrated below:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  BM1, BM2: TBookmarkStr;
begin
  with ADODataSet1 do begin
    BM1 := Bookmark;
    BMList.Add(Pointer(BM1));
    MoveBy(3);
    BM2 := Bookmark;
    BMList.Add(Pointer(BM2));
    FilterOnBookmarks([BM1, BM2]);
  end;
end;
```

Note that the example above also adds the bookmarks to a list object named BMList. This is necessary so that the application can later free the bookmarks when they are no longer needed.

For details on using bookmarks, see "Marking and returning to records" on page 24-9. For details on other types of filters, see "Displaying and editing a subset of data using filters" on page 24-13.

## Fetching records asynchronously

Unlike other datasets, ADO datasets can fetch their data asynchronously. This allows your application to continue performing other tasks while the dataset populates itself with data from the data store.

To control whether the dataset fetches data asynchronously, if it fetches data at all, use the *ExecuteOptions* property. *ExecuteOptions* governs how the dataset fetches its records when you call *Open* or set *Active* to *True*. If the dataset represents a query or stored procedure that does not return any records, ExecuteOptions governs how the query or stored procedure is executed when you call *ExecSQL* or *ExecProc*.

*ExecuteOptions* is a set that includes zero or more of the following values:

**Table 27.4** Execution options for ADO datasets

| Execute Option | Meaning |
|---|---|
| eoAsyncExecute | The command or data fetch operation is executed asynchronously. |
| eoAsyncFetch | The dataset first fetches the number of records specified by the *CacheSize* property synchronously, then fetches any remaining rows asynchronously. |
| eoAsyncFetchNonBlocking | Asynchronous data fetches or command execution do not block the current thread of execution. |
| eoExecuteNoRecords | A command or stored procedure that does not return data. If any rows are retrieved, they are discarded and not returned. |

## Using batch updates

One approach for caching updates is to connect the ADO dataset to a client dataset using a dataset provider. This approach is discussed in "Using a client dataset to cache updates" on page 29-16.

However, ADO dataset components provide their own support for cached updates, which they call batch updates. The following table lists the correspondences between caching updates using a client dataset and using the batch updates features:

**Table 27.5** Comparison of ADO and client dataset cached updates

| ADO dataset | TClientDataSet | Description |
|---|---|---|
| LockType | Not used: client datasets always cache updates | Specifies whether the dataset is opened in batch update mode. |
| CursorType | Not used: client datasets always work with an in-memory snapshot of data | Specifies how isolated the ADO dataset is from changes on the server. |
| RecordStatus | UpdateStatus | Indicates what update, if any, has occurred on the current row. *RecordStatus* provides more information than *UpdateStatus*. |
| FilterGroup | StatusFilter | Specifies which type of records are available. *FilterGroup* provides a wider variety of information. |
| UpdateBatch | ApplyUpdates | Applies the cached updates back to the database server. Unlike *ApplyUpdates*, *UpdateBatch* lets you limit the types of updates to be applied. |
| CancelBatch | CancelUpdates | Discards pending updates, reverting to the original values. Unlike *CancelUpdates*, *CancelBatch* lets you limit the types of updates to be canceled. |

Using the batch updates features of ADO dataset components is a matter of:

- Opening the dataset in batch update mode
- Inspecting the update status of individual rows
- Filtering multiple rows based on update status
- Applying the batch updates to base tables
- Canceling batch updates

### Opening the dataset in batch update mode

To open an ADO dataset in batch update mode, it must meet these criteria:

1. The component's *CursorType* property must be *ctKeySet* (the default property value) or *ctStatic*.
2. The *LockType* property must be *ltBatchOptimistic*.
3. The command must be a SELECT query.

Before activating the dataset component, set the *CursorType* and *LockType* properties as indicated above. Assign a SELECT statement to the component's *CommandText* property (for *TADODataSet*) or the *SQL* property (for *TADOQuery*). For *TADOStoredProc* components, set the *ProcedureName* to the name of a stored procedure that returns a result set. These properties can be set at design-time through the Object Inspector or programmatically at runtime. The example below shows the preparation of a *TADODataSet* component for batch update mode.

```
with ADODataSet1 do begin
  CursorLocation := clUseClient;
  CursorType := ctStatic;
  LockType := ltBatchOptimistic;
  CommandType := cmdText;
  CommandText := 'SELECT * FROM Employee';
  Open;
end;
```

After a dataset has been opened in batch update mode, all changes to the data are cached rather than applied directly to the base tables.

### Inspecting the update status of individual rows

Determine the update status of a given row by making it current and then inspecting the *RecordStatus* property of the ADO data component. *RecordStatus* reflects the update status of the current row and only that row.

```
case ADOQuery1.RecordStatus of
  rsUnmodified: StatusBar1.Panels[0].Text := 'Unchanged record';
  rsModified:   StatusBar1.Panels[0].Text := 'Changed record';
  rsDeleted:    StatusBar1.Panels[0].Text := 'Deleted record';
  rsNew:        StatusBar1.Panels[0].Text := 'New record';
end;
```

### Filtering multiple rows based on update status

Filter a recordset to show only those rows that belong to a group of rows with the same update status using the *FilterGroup* property. Set *FilterGroup* to the *TFilterGroup* constant that represents the update status of rows to display. A value of *fgNone* (the default value for this property) specifies that no filtering is applied and all rows are visible regardless of update status (except rows marked for deletion). The example below causes only pending batch update rows to be visible.

```
FilterGroup := fgPendingRecords;
Filtered := True;
```

**Note** For the *FilterGroup* property to have an effect, the ADO dataset component's *Filtered* property must be set to *True*.

### Applying the batch updates to base tables

Apply pending data changes that have not yet been applied or canceled by calling the *UpdateBatch* method. Rows that have been changed and are applied have their changes put into the base tables on which the recordset is based. A cached row marked for deletion causes the corresponding base table row to be deleted. A record insertion (exists in the cache but not the base table) is added to the base table. Modified rows cause the columns in the corresponding rows in the base tables to be changed to the new column values in the cache.

Used alone with no parameter, *UpdateBatch* applies all pending updates. A *TAffectRecords* value can optionally be passed as the parameter for *UpdateBatch*. If any value except *arAll* is passed, only a subset of the pending changes are applied. Passing *arAll* is the same as passing no parameter at all and causes all pending updates to be applied. The example below applies only the currently active row to be applied:

```
ADODataSet1.UpdateBatch(arCurrent);
```

### Canceling batch updates

Cancel pending data changes that have not yet been canceled or applied by calling the *CancelBatch* method. When you cancel pending batch updates, field values on rows that have been changed revert to the values that existed prior to the last call to *CancelBatch* or *UpdateBatch*, if either has been called, or prior to the current pending batch of changes.

Used alone with no parameter, *CancelBatch* cancels all pending updates. A *TAffectRecords* value can optionally be passed as the parameter for *CancelBatch*. If any value except *arAll* is passed, only a subset of the pending changes are canceled.Passing *arAll* is the same as passing no parameter at all and causes all pending updates to be canceled. The example below cancels all pending changes:

```
ADODataSet1.CancelBatch;
```

## Loading data from and saving data to files

The data retrieved via an ADO dataset component can be saved to a file for later retrieval on the same or a different computer. The data is saved in one of two proprietary formats: ADTG or XML. These two file formats are the only formats supported by ADO. However, both formats are not necessarily supported in all versions of ADO. Consult the ADO documentation for the version you are using to determine what save file formats are supported.

Save the data to a file using the *SaveToFile* method. *SaveToFile* takes two parameters, the name of the file to which data is saved, and, optionally, the format (ADTG or XML) in which to save the data. Indicate the format for the saved file by setting the *Format* parameter to *pfADTG* or *pfXML*. If the file specified by the *FileName* parameter already exists, *SaveToFile* raises an *EOleException*.

Retrieve the data from file using the *LoadFromFile* method. *LoadFromFile* takes a single parameter, the name of the file to load. If the specified file does not exist, *LoadFromFile* raises an *EOleException* exception. On calling the *LoadFromFile* method, the dataset component is automatically activated.

In the example below, the first procedure saves the dataset retrieved by the *TADODataSet* component *ADODataSet1* to a file. The target file is an ADTG file named SaveFile, saved to a local drive. The second procedure loads this saved file into the *TADODataSet* component *ADODataSet2*.

```
procedure TForm1.SaveBtnClick(Sender: TObject);
begin
  if (FileExists('c:\SaveFile')) then
  begin
    DeleteFile('c:\SaveFile');
    StatusBar1.Panels[0].Text := 'Save file deleted!';
  end;
  ADODataSet1.SaveToFile('c:\SaveFile', pfADTG);
end;

procedure TForm1.LoadBtnClick(Sender: TObject);
begin
  if (FileExists('c:\SaveFile')) then
    ADODataSet2.LoadFromFile('c:\SaveFile')
  else
    StatusBar1.Panels[0].Text := 'Save file does not exist!';
end;
```

The datasets that save and load the data need not be on the same form as above, in the same application, or even on the same computer. This allows for the briefcase-style transfer of data from one computer to another.

## Using TADODataSet

*TADODataSet* is a general-purpose dataset for working with data from an ADO data store. Unlike the other ADO dataset components, *TADODataSet* is not a table-type, query-type, or stored procedure-type dataset. Instead, it can function as any of these types:

- Like a table-type dataset, *TADODataSet* lets you represent all of the rows and columns of a single database table. To use it in this way, set the *CommandType* property to *cmdTable* and the *CommandText* property to the name of the table. *TADODataSet* supports table-type tasks such as
  - Assigning indexes to sort records or form the basis of record-based searches. In addition to the standard index properties and methods described in "Sorting records with indexes" on page 24-26, *TADODataSet* lets you sort using temporary indexes by setting the *Sort* property. Indexed-based searches performed using the *Seek* method use the current index.
  - Emptying the dataset. The *DeleteRecords* method provides greater control than related methods in other table-type datasets, because it lets you specify what records to delete.

  The table-type tasks supported by *TADODataSet* are available even when you are not using a *CommandType* of *cmdTable*.

- Like a query-type dataset, *TADODataSet* lets you specify a single SQL command that is executed when you open the dataset. To use it in this way, set the *CommandType* property to *cmdText* and the *CommandText* property to the SQL command you want to execute. At design time, you can double-click on the *CommandText* property in the Object Inspector to use the Command Text editor for help in constructing the SQL command. *TADODataSet* supports query-type tasks such as
  - Using parameters in the query text. See "Using parameters in queries" on page 24-45 for details on query parameters.
  - Setting up master/detail relationships using parameters. See "Establishing master/detail relationships using parameters" on page 24-47 for details on how to do this.
  - Preparing the query in advance to improve performance by setting the *Prepared* property to *True*.
- Like a stored procedure-type dataset, *TADODataSet* lets you specify a stored procedure that is executed when you open the dataset. To use it in this way, set the *CommandType* property to *cmdStoredProc* and the *CommandText* property to the name of the stored procedure. *TADODataSet* supports stored procedure-type tasks such as
  - Working with stored procedure parameters. See "Working with stored procedure parameters" on page 24-51 for details on stored procedure parameters.
  - Fetching multiple result sets. See "Fetching multiple result sets" on page 24-56 for details on how to do this.
  - Preparing the stored procedure in advance to improve performance by setting the *Prepared* property to *True*.

In addition, *TADODataSet* lets you work with data stored in files by setting the *CommandType* property to *cmdFile* and the *CommandText* property to the file name.

Before you set the *CommandText* and *CommandType* properties, you should link the *TADODataSet* to a data store by setting the *Connection* or *ConnectionString* property. This process is described in "Connecting an ADO dataset to a data store" on page 27-10. As an alternative, you can use an RDS DataSpace object to connect the *TADODataSet* to an ADO-based application server. To use an RDS DataSpace object, set the *RDSConnection* property to a *TRDSConnection* object.

## Using Command objects

In the ADO environment, commands are textual representations of provider-specific action requests. Typically, they are Data Definition Language (DDL) and Data Manipulation Language (DML) SQL statements. The language used in commands is provider-specific, but usually compliant with the SQL-92 standard for the SQL language.

Although you can always execute commands using *TADOQuery*, you may not want the overhead of using a dataset component, especially if the command does not return a result set. As an alternative, you can use the *TADOCommand* component, which is a lighter-weight object designed to execute commands, one command at a time. *TADOCommand* is intended primarily for executing those commands that do not return result sets, such as Data Definition Language (DDL) SQL statements. Through an overloaded version of its *Execute* method, however, it is capable of returning a result set that can be assigned to the *RecordSet* property of an ADO dataset component.

In general, working with *TADOCommand* is very similar to working with *TADODataSet*, except that you can't use the standard dataset methods to fetch data, navigate records, edit data, and so on. *TADOCommand* objects connect to a data store in the same way as ADO datasets. See "Connecting an ADO dataset to a data store" on page 27-10 for details.

The following topics provide details on how to specify and execute commands using *TADOCommand*.

### Specifying the command

Specify commands for a *TADOCommand* component using the *CommandText* property. Like *TADODataSet*, *TADOCommand* lets you specify the command in different ways, depending on the *CommandType* property. Possible values for *CommandType* include: *cmdText* (used if the command is an SQL statement), *cmdTable* (if it is a table name), and *cmdStoredProc* (if the command is the name of a stored procedure). At design-time, select the appropriate command type from the list in the Object Inspector. At runtime, assign a value of type *TCommandType* to the *CommandType* property.

```
with ADOCommand1 do begin
  CommandText := 'AddEmployee';
  CommandType := cmdStoredProc;
ƒ
end;
```

If no specific type is specified, the server is left to decide as best it can based on the command in *CommandText*.

*CommandText* can contain the text of an SQL query that includes parameters or the name of a stored procedure that uses parameters. You must then supply parameter values, which are bound to the parameters before executing the command. See "Handling command parameters" on page 27-20 for details.

## Using the Execute method

Before *TADOCommand* can execute its command, it must have a valid connection to a data store. This is established just as with an ADO dataset. See "Connecting an ADO dataset to a data store" on page 27-10 for details.

To execute the command, call the *Execute* method. *Execute* is an overloaded method that lets you choose the most appropriate way to execute the command.

For commands that do not require any parameters and for which you do not need to know how many records were affected, call *Execute* without any parameters:

```
with ADOCommand1 do begin
  CommandText := 'UpdateInventory';
  CommandType := cmdStoredProc;
  Execute;
end;
```

Other versions of *Execute* let you provide parameter values using a Variant array, and to obtain the number of records affected by the command.

For information on executing commands that return a result set, see "Retrieving result sets with commands" on page 27-20.

## Canceling commands

If you are executing the command asynchronously, then after calling *Execute* you can abort the execution by calling the *Cancel* method:

```
procedure TDataForm.ExecuteButtonClick(Sender: TObject);
begin
  ADOCommand1.Execute;
end;
procedure TDataForm.CancelButtonClick(Sender: TObject);
begin
  ADOCommand1.Cancel;
end;
```

The *Cancel* method only has an effect if there is a command pending and it was executed asynchronously (*eoAsynchExecute* is in the *ExecuteOptions* parameter of the *Execute* method). A command is said to be pending if the *Execute* method has been called but the command has not yet been completed or timed out.

A command times out if it is not completed or canceled before the number of seconds specified in the *CommandTimeout* property expire. By default, commands time out after 30 seconds.

## Retrieving result sets with commands

Unlike *TADOQuery* components, which use different methods to execute depending on whether they return a result set, *TADOCommand* always uses the *Execute* command to execute the command, regardless of whether it returns a result set. When the command returns a result set, *Execute* returns an interface to the ADO _RecordSet interface.

The most convenient way to work with this interface is to assign it to the *RecordSet* property of an ADO dataset.

For example, the following code uses *TADOCommand* (*ADOCommand1*) to execute a SELECT query, which returns a result set. This result set is then assigned to the *RecordSet* property of a *TADODataSet* component (*ADODataSet1*).

```
with ADOCommand1 do begin
  CommandText := 'SELECT Company, State ' +
    'FROM customer ' +
    'WHERE State = :StateParam'; CommandType :=
  cmdText; Parameters.ParamByName('StateParam').Value
  := 'HI'; ADODataSet1.Recordset := Execute;
end;
```

As soon as the result set is assigned to the ADO dataset's *Recordset* property, the dataset is automatically activated and the data is available.

## Handling command parameters

There are two ways in which a *TADOCommand* object may use parameters:

- The *CommandText* property can specify a query that includes parameters. Working with parameterized queries in *TADOCommand* works like using a parameterized query in an ADO dataset. See "Using parameters in queries" on page 24-45 for details on parameterized queries.
- The *CommandText* property can specify a stored procedure that uses parameters. Stored procedure parameters work much the same using *TADOCommand* as with an ADO dataset. See "Working with stored procedure parameters" on page 24-51 for details on stored procedure parameters.

There are two ways to supply parameter values when working with *TADOCommand*: you can supply them when you call the *Execute* method, or you can specify them ahead of time using the *Parameters* property.

The *Execute* method is overloaded to include versions that take a set of parameter values as a Variant array. This is useful when you want to supply parameter values quickly without the overhead of setting up the *Parameters* property:

```
ADOCommand1.Execute(VarArrayOf([Edit1.Text, Date]));
```

When working with stored procedures that return output parameters, you must use the *Parameters* property instead. Even if you do not need to read output parameters, you may prefer to use the *Parameters* property, which lets you supply parameters at design time and lets you work with *TADOCommand* properties in the same way you work with the parameters on datasets.

When you set the *CommandText* property, the *Parameters* property is automatically updated to reflect the parameters in the query or those used by the stored procedure. At design-time, you can use the Parameter Editor to access parameters, by clicking the ellipsis button for the *Parameters* property in the Object Inspector. At runtime, use properties and methods of *TParameter* to set (or get) the values of each parameter.

```
with ADOCommand1 do begin
  CommandText := 'INSERT INTO Talley ' +
    '(Counter) ' +
    'VALUES (:NewValueParam)';
  CommandType := cmdText;
  Parameters.ParamByName('NewValueParam').Value := 57;
  Execute
end;
```

Chapter

28

# Using unidirectional datasets

*dbExpress* is a set of lightweight database drivers that provide fast access to SQL database servers. For each supported database, *dbExpress* provides a driver that adapts the server-specific software to a set of uniform *dbExpress* interfaces. When you deploy a database application that uses *dbExpress*, you need only include a dll (the server-specific driver) with the application files you build.

*dbExpress* lets you access databases using unidirectional datasets. Unidirectional datasets are designed for quick lightweight access to database information, with minimal overhead. Like other datasets, they can send an SQL command to the database server, and if the command returns a set of records, obtain a cursor for accessing those records. However, unidirectional datasets can only retrieve a unidirectional cursor. They do not buffer data in memory, which makes them faster and less resource-intensive than other types of dataset. However, because there are no buffered records, unidirectional datasets are also less flexible than other datasets. Many of the capabilities introduced by *TDataSet* are either unimplemented in unidirectional datasets, or cause them to raise exceptions. For example:

- The only supported navigation methods are the *First* and *Next* methods. Most others raise exceptions. Some, such as the methods involved in bookmark support, simply do nothing.
- There is no built-in support for editing because editing requires a buffer to hold the edits. The *CanModify* property is always *False*, so attempts to put the dataset into edit mode always fail. You can, however, use unidirectional datasets to update data using an SQL UPDATE command or provide conventional editing support by using a dbExpress-enabled client dataset or connecting the dataset to a client dataset (see “Connecting to another dataset” on page 19-10).

- There is no support for filters, because filters work with multiple records, which requires buffering. If you try to filter a unidirectional dataset, it raises an exception. Instead, all limits on what data appears must be imposed using the SQL command that defines the data for the dataset.
- There is no support for lookup fields, which require buffering to hold multiple records containing lookup values. If you define a lookup field on a unidirectional dataset, it does not work properly.

Despite these limitations, unidirectional datasets are a powerful way to access data. They are the fastest data access mechanism, and very simple to use and deploy.

## Types of unidirectional datasets

The *dbExpress* page of the Component palette contains four types of unidirectional dataset: *TSQLDataSet, TSQLQuery, TSQLTable,* and *TSQLStoredProc.*

*TSQLDataSet* is the most general of the four. You can use an SQL dataset to represent any data available through *dbExpress*, or to send commands to a database accessed through *dbExpress*. This is the recommended component to use for working with database tables in new database applications.

*TSQLQuery* is a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any. See “Using query-type datasets” on page 24-42 for information on using query-type datasets.

*TSQLTable* is a table-type dataset that represents all of the rows and columns of a single database table. See “Using table type datasets” on page 24-25 for information on using table-type datasets.

*TSQLStoredProc* is a stored procedure-type dataset that executes a stored procedure defined on a database server. See “Using stored procedure-type datasets” on page 24-50 for information on using stored procedure-type datasets.

**Note** The *dbExpress* page also includes *TSimpleDataSet*, which is not a unidirectional dataset. Rather, it is a client dataset that uses a unidirectional dataset internally to access its data.

## Connecting to the database server

The first step when working with a unidirectional dataset is to connect it to a database server. At design time, once a dataset has an active connection to a database server, the Object Inspector can provide drop-down lists of values for other properties. For example, when representing a stored procedure, you must have an active connection before the Object Inspector can list what stored procedures are available on the server.

The connection to a database server is represented by a separate *TSQLConnection* component. You work with *TSQLConnection* like any other database connection component. For information about database connection components, see Chapter 23, "Connecting to databases."

To use *TSQLConnection* to connect a unidirectional dataset to a database server, set the *SQLConnection* property. At design time, you can choose the SQL connection component from a drop-down list in the Object Inspector. If you make this assignment at runtime, be sure that the connection is active:

```
SQLDataSet1.SQLConnection := SQLConnection1;
SQLConnection1.Connected := True;
```

Typically, all unidirectional datasets in an application share the same connection component, unless you are working with data from multiple database servers. However, you may want to use a separate connection for each dataset if the server does not support multiple statements per connection. Check whether the database server requires a separate connection for each dataset by reading the *MaxStmtsPerConn* property. By default, *TSQLConnection* generates connections as needed when the server limits the number of statements that can be executed over a connection. If you want to keep stricter track of the connections you are using, set the *AutoClone* property to *False*.

Before you assign the *SQLConnection* property, you will need to set up the *TSQLConnection* component so that it identifies the database server and any required connection parameters (including which database to use on the server, the host name of the machine running the server, the username, password, and so on).

## Setting up TSQLConnection

In order to describe a database connection in sufficient detail for *TSQLConnection* to open a connection, you must identify both the driver to use and a set of connection parameters the are passed to that driver.

### Identifying the driver

The driver is identified by the *DriverName* property, which is the name of an installed *dbExpress* driver, such as INTERBASE, INFORMIX, ORACLE, MYSQL, MSSQL, or DB2. The driver name is associated with two files:

- The *dbExpress* driver. This can be either a dynamic-link library with a name like dbexpint.dll, dbexpora.dll, dbexpmysql.dll, dbexpmss.dll, or dbexpdb2.dll, or a compiled unit that you can statically link into your application (dbexpint.dcu, dbexpora.dcu, dbexpmys.dcu, dbexpmss.dcu, or dbexpdb2.dcu).
- The dynamic-link library provided by the database vendor for client-side support.

The relationship between these two files and the database name is stored in a file called dbxdrivers.ini, which is updated when you install a *dbExpress* driver. Typically, you do not need to worry about these files because the SQL connection component looks them up in dbxdrivers.ini when given the value of *DriverName*. When you set the *DriverName* property, *TSQLConnection* automatically sets the *LibraryName* and *VendorLib* properties to the names of the associated dlls. Once *LibraryName* and *VendorLib* have been set, your application does not need to rely on dbxdrivers.ini. (That is, you do not need to deploy dbxdrivers.ini with your application unless you set the *DriverName* property at runtime.)

### Specifying connection parameters

The *Params* property is a string list that lists name/value pairs. Each pair has the form *Name=Value*, where *Name* is the name of the parameter, and *Value* is the value you want to assign.

The particular parameters you need depend on the database server you are using. However, one particular parameter, *Database*, is required for all servers. Its value depends on the server you are using. For example, with InterBase, *Database* is the name of the .gdb file, with ORACLE it is the entry in TNSNames.ora, while with DB2, it is the client-side node name.

Other typical parameters include the *User_Name* (the name to use when logging in), *Password* (the password for *User_Name*), *HostName* (the machine name or IP address of where the server is located), and *TransIsolation* (the degree to which transactions you introduce are aware of changes made by other transactions). When you specify a driver name, the *Params* property is preloaded with all the parameters you need for that driver type, initialized to default values.

Because *Params* is a string list, at design time you can double-click on the *Params* property in the Object Inspector to edit the parameters using the String List editor. At runtime, use the *Params.Values* property to assign values to individual parameters.

### Naming a connection description

Although you can always specify a connection using only the *DatabaseName* and *Params* properties, it can be more convenient to name a specific combination and then just identify the connection by name. You can name *dbExpress* database and parameter combinations, which are then saved in a file called dbxconnections.ini. The name of each combination is called a connection name.

Once you have defined the connection name, you can identify a database connection by simply setting the *ConnectionName* property to a valid connection name. Setting *ConnectionName* automatically sets the *DriverName* and *Params* properties. Once *ConnectionName* is set, you can edit the *Params* property to create temporary differences from the saved set of parameter values, but changing the *DriverName* property clears both *Params* and *ConnectionName*.

One advantage of using connection names arises when you develop your application using one database (for example Local InterBase), but deploy it for use with another (such as ORACLE). In that case, *DriverName* and *Params* will likely differ on the system where you deploy your application from the values you use during development. You can switch between the two connection descriptions easily by using two versions of the dbxconnections.ini file. At design-time, your application loads the *DriverName* and *Params* from the design-time version of dbxconnections.ini. Then, when you deploy your application, it loads these values from a separate version of dbxconnections.ini that uses the "real" database. However, for this to work, you must instruct your connection component to reload the *DriverName* and *Params* properties at runtime. There are two ways to do this:

- Set the *LoadParamsOnConnect* property to *True*. This causes *TSQLConnection* to automatically set *DriverName* and *Params* to the values associated with *ConnectionName* in dbxconnections.ini when the connection is opened.
- Call the *LoadParamsFromIniFile* method. This method sets *DriverName* and *Params* to the values associated with *ConnectionName* in dbxconnections.ini (or in another file that you specify). You might choose to use this method if you want to then override certain parameter values before opening the connection.

### Using the Connection Editor

The relationships between connection names and their associated driver and connection parameters is stored in the dbxconnections.ini file. You can create or modify these associations using the Connection Editor.

To display the Connection Editor, double-click on the *TSQLConnection* component. The Connection Editor appears, with a drop-down list containing all available drivers, a list of connection names for the currently selected driver, and a table listing the connection parameters for the currently selected connection name.

You can use this dialog to indicate the connection to use by selecting a driver and connection name. Once you have chosen the configuration you want, click the Test Connection button to check that you have chosen a valid configuration.

In addition, you can use this dialog to edit the named connections in dbxconnections.ini:

- Edit the parameter values in the parameter table to change the currently selected named connection. When you exit the dialog by clicking OK, the new parameter values are saved to dbxconnections.ini.
- Click the Add Connection button to define a new named connection. A dialog appears where you specify the driver to use and the name of the new connection. Once the connection is named, edit the parameters to specify the connection you want and click the OK button to save the new connection to dbxconnections.ini.
- Click the Delete Connection button to delete the currently selected named connection from dbxconnections.ini.
- Click the Rename Connection button to change the name of the currently selected named connection. Note that any edits you have made to the parameters are saved with the new name when you click the OK button.

## Specifying what data to display

There are a number of ways to specify what data a unidirectional dataset represents. Which method you choose depends on the type of unidirectional dataset you are using and whether the information comes from a single database table, the results of a query, or from a stored procedure.

When you work with a *TSQLDataSet* component, use the *CommandType* property to indicate where the dataset gets its data. *CommandType* can take any of the following values:

- *ctQuery*: When *CommandType* is *ctQuery*, *TSQLDataSet* executes a query you specify. If the query is a SELECT command, the dataset contains the resulting set of records.
- *ctTable*: When *CommandType* is *ctTable*, *TSQLDataSet* retrieves all of the records from a specified table.
- *ctStoredProc*: When *CommandType* is *ctStoredProc*, *TSQLDataSet* executes a stored procedure. If the stored procedure returns a cursor, the dataset contains the returned records.

**Note** You can also populate the unidirectional dataset with metadata about what is available on the server. For information on how to do this, see "Fetching metadata into a unidirectional dataset" on page 28-13.

### Representing the results of a query

Using a query is the most general way to specify a set of records. Queries are simply commands written in SQL. You can use either *TSQLDataSet* or *TSQLQuery* to represent the result of a query.

When using *TSQLDataSet*, set the *CommandType* property to *ctQuery* and assign the text of the query statement to the *CommandText* property. When using *TSQLQuery*, assign the query to the *SQL* property instead. These properties work the same way for all general-purpose or query-type datasets. "Specifying the query" on page 24-43 discusses them in greater detail.

When you specify the query, it can include parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values that appear in the SQL statement. Using parameters in queries and supplying values for those parameters is discussed in "Using parameters in queries" on page 24-45.

SQL defines queries such as UPDATE queries that perform actions on the server but do not return records. Such queries are discussed in "Executing commands that do not return records" on page 28-10.

## Representing the records in a table

When you want to represent all of the fields and all of the records in a single underlying database table, you can use either *TSQLDataSet* or *TSQLTable* to generate the query for you rather than writing the SQL yourself.

**Note** If server performance is a concern, you may want to compose the query explicitly rather than relying on an automatically-generated query. Automatically-generated queries use wildcards rather than explicitly listing all of the fields in the table. This can result in slightly slower performance on the server. The wildcard (*) in automatically-generated queries is more robust to changes in the fields on the server.

### Representing a table using TSQLDataSet

To make *TSQLDataSet* generate a query to fetch all fields and all records of a single database table, set the *CommandType* property to *ctTable*.

When *CommandType* is *ctTable*, *TSQLDataSet* generates a query based on the values of two properties:

- *CommandText* specifies the name of the database table that the *TSQLDataSet* object should represent.
- *SortFieldNames* lists the names of any fields to use to sort the data, in the order of significance.

For example, if you specify the following:

```
SQLDataSet1.CommandType := ctTable;
SQLDataSet1.CommandText := 'Employee';
SQLDataSet1.SortFieldNames := 'HireDate,Salary'
```

*TSQLDataSet* generates the following query, which lists all the records in the Employee table, sorted by HireDate and, within HireDate, by Salary:

```
select * from Employee order by HireDate, Salary
```

### Representing a table using TSQLTable

When using *TSQLTable*, specify the table you want using the *TableName* property.

To specify the order of fields in the dataset, you must specify an index. There are two ways to do this:

- Set the *IndexName* property to the name of an index defined on the server that imposes the order you want.
- Set the *IndexFieldNames* property to a semicolon-delimited list of field names on which to sort. *IndexFieldNames* works like the *SortFieldNames* property of *TSQLDataSet*, except that it uses a semicolon instead of a comma as a delimiter.

## Representing the results of a stored procedure

Stored procedures are sets of SQL statements that are named and stored on an SQL server. How you indicate the stored procedure you want to execute depends on the type of unidirectional dataset you are using.

When using *TSQLDataSet*, to specify a stored procedure:

- Set the *CommandType* property to *ctStoredProc*.
- Specify the name of the stored procedure as the value of the *CommandText* property:

```
SQLDataSet1.CommandType := ctStoredProc;
SQLDataSet1.CommandText := 'MyStoredProcName';
```

When using *TSQLStoredProc*, you need only specify the name of the stored procedure as the value of the *StoredProcName* property.

```
SQLStoredProc1.StoredProcName := 'MyStoredProcName';
```

After you have identified a stored procedure, your application may need to enter values for any input parameters of the stored procedure or retrieve the values of output parameters after you execute the stored procedure. See “Working with stored procedure parameters” on page 24-51 for information about working with stored procedure parameters.

## Fetching the data

Once you have specified the source of the data, you must fetch the data before your application can access it. Once the dataset has fetched the data, data-aware controls linked to the dataset through a data source automatically display data values and client datasets linked to the dataset through a provider can be populated with records.

As with any dataset, there are two ways to fetch the data for a unidirectional dataset:

- Set the *Active* property to *True*, either at design time in the Object Inspector, or in code at runtime:

```
    CustQuery.Active := True;
```

- Call the *Open* method at runtime,

```
    CustQuery.Open;
```

Use the *Active* property or the *Open* method with any unidirectional dataset that obtains records from the server. It does not matter whether these records come from a SELECT query (including automatically-generated queries when the *CommandType* is *ctTable*) or a stored procedure.

## Preparing the dataset

Before a query or stored procedure can execute on the server, it must first be "prepared". Preparing the dataset means that *dbExpress* and the server allocate resources for the statement and its parameters. If *CommandType* is *ctTable*, this is when the dataset generates its SELECT query. Any parameters that are not bound by the server are folded into a query at this point.

Unidirectional datasets are automatically prepared when you set *Active* to *True* or call the *Open* method. When you close the dataset, the resources allocated for executing the statement are freed. If you intend to execute the query or stored procedure more than once, you can improve performance by explicitly preparing the dataset before you open it the first time. To explicitly prepare a dataset, set its *Prepared* property to *True*.

```
CustQuery.Prepared := True;
```

When you explicitly prepare the dataset, the resources allocated for executing the statement are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you change a parameter value or the *SortFieldNames* property).

## Fetching multiple datasets

Some stored procedures return multiple sets of records. The dataset only fetches the first set when you open it. In order to access the other sets of records, call the *NextRecordSet* method:

```
var
  DataSet2: TCustomSQLDataSet;
  nRows: Integer;
begin
  DataSet2 := SQLStoredProc1.NextRecordSet;
  ƒ
```

*NextRecordSet* returns a newly created *TCustomSQLDataSet* component that provides access to the next set of records. That is, the first time you call *NextRecordSet*, it returns a dataset for the second set of records. Calling *NextRecordSet* returns a third dataset, and so on, until there are no more sets of records. When there are no additional datasets, *NextRecordSet* returns **nil**.

## Executing commands that do not return records

You can use a unidirectional dataset even if the query or stored procedure it represents does not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language.

The SQL command you execute must be acceptable to the server you are using. Unidirectional datasets neither evaluate the SQL nor execute it. They merely pass the command to the server for execution.

**Note** If the command does not return any records, you do not need to use a unidirectional dataset at all, because there is no need for the dataset methods that provide access to a set of records. The SQL connection component that connects to the database server can be used directly to execute a command on the server. See "Sending commands to the server" on page 23-10 for details.

### Specifying the command to execute

With unidirectional datasets, the way you specify the command to execute is the same whether the command results in a dataset or not. That is:

When using *TSQLDataSet*, use the *CommandType* and *CommandText* properties to specify the command:

- If *CommandType* is *ctQuery*, *CommandText* is the SQL statement to pass to the server.
- If *CommandType* is *ctStoredProc*, *CommandText* is the name of a stored procedure to execute.

When using *TSQLQuery*, use the *SQL* property to specify the SQL statement to pass to the server.

When using *TSQLStoredProc*, use the *StoredProcName* property to specify the name of the stored procedure to execute.

Just as you specify the command in the same way as when you are retrieving records, you work with query parameters or stored procedure parameters the same way as with queries and stored procedures that return records. See "Using parameters in queries" on page 24-45 and "Working with stored procedure parameters" on page 24-51 for details.

## Executing the command

To execute a query or stored procedure that does not return any records, you do not use the *Active* property or the *Open* method. Instead, you must use

- The *ExecSQL* method if the dataset is an instance of *TSQLDataSet* or *TSQLQuery*.

```
FixTicket.CommandText := 'DELETE FROM TrafficViolations WHERE (TicketID = 1099)';
FixTicket.ExecSQL;
```

- The *ExecProc* method if the dataset is an instance of *TSQLStoredProc*.

```
SQLStoredProc1.StoredProcName := 'MyCommandWithNoResults';
SQLStoredProc1.ExecProc;
```

**Tip** If you are executing the query or stored procedure multiple times, it is a good idea to set the *Prepared* property to *True*.

## Creating and modifying server metadata

Most of the commands that do not return data fall into two categories: those that you use to edit data (such as INSERT, DELETE, and UPDATE commands), and those that you use to create or modify entities on the server such as tables, indexes, and stored procedures.

If you don't want to use explicit SQL commands for editing, you can link your unidirectional dataset to a client dataset and let it handle all the generation of all SQL commands concerned with editing (see "Connecting a client dataset to another dataset in the same application" on page 19-12). In fact, this is the recommended approach because data-aware controls are designed to perform edits through a dataset such as *TClientDataSet*.

The only way your application can create or modify metadata on the server, however, is to send a command. Not all database drivers support the same SQL syntax. It is beyond the scope of this topic to describe the SQL syntax supported by each database type and the differences between the database types. For a comprehensive and up-to-date discussion of the SQL implementation for a given database system, see the documentation that comes with that system.

In general, use the CREATE TABLE statement to create tables in a database and CREATE INDEX to create new indexes for those tables. Where supported, use other CREATE statements for adding various metadata objects, such as CREATE DOMAIN, CREATE VIEW, CREATE SCHEMA, and CREATE PROCEDURE.

For each of the CREATE statements, there is a corresponding DROP statement to delete the metadata object. These statements include DROP TABLE, DROP VIEW, DROP DOMAIN, DROP SCHEMA, and DROP PROCEDURE.

To change the structure of a table, use the ALTER TABLE statement. ALTER TABLE has ADD and DROP clauses to create new elements in a table and to delete them. For example, use the ADD COLUMN clause to add a new column to the table and DROP CONSTRAINT to delete an existing constraint from the table.

For example, the following statement creates a stored procedure called GET_EMP_PROJ on an InterBase database:

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

The following code uses a *TSQLDataSet* to create this stored procedure. Note the use of the *ParamCheck* property to prevent the dataset from confusing the parameters in the stored procedure definition (:EMP_NO and :PROJ_ID) with a parameter of the query that creates the stored procedure.

```
with SQLDataSet1 do
begin
  ParamCheck := False;
  CommandType := ctQuery;
  CommandText := 'CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT) ' +
      'RETURNS (PROJ_ID CHAR(5)) AS ' +
      'BEGIN ' +
        'FOR SELECT PROJ_ID FROM EMPLOYEE_PROJECT ' +
        'WHERE EMP_NO = :EMP_NO ' +
        'INTO :PROJ_ID ' +
          'DO SUSPEND; ' +
      'END';
  ExecSQL;
end;
```

## Setting up master/detail linked cursors

There are two ways to use linked cursors to set up a master/detail relationship with a unidirectional dataset as the detail set. Which method you use depends on the type of unidirectional dataset you are using. Once you have set up such a relationship, the unidirectional dataset (the “many” in a one-to-many relationship) provides access only to those records that correspond to the current record on the master set (the “one” in the one-to-many relationship).

*TSQLDataSet* and *TSQLQuery* require you to use a parameterized query to establish a master/detail relationship. This is the technique for creating such relationships on all query-type datasets. For details on creating master/detail relationships with query-type datasets, see “Establishing master/detail relationships using parameters” on page 24-47.

To set up a master/detail relationship where the detail set is an instance of *TSQLTable*, use the *MasterSource* and *MasterFields* properties, just as you would with any other table-type dataset. For details on creating master/detail relationships with table-type datasets, see "Establishing master/detail relationships using parameters" on page 24-47.

## Accessing schema information

There are two ways to obtain information about what is available on the server. This information, called schema information or metadata, includes information about what tables and stored procedures are available on the server and information about these tables and stored procedures (such as the fields a table contains, the indexes that are defined, and the parameters a stored procedure uses).

The simplest way to obtain this metadata is to use the methods of *TSQLConnection*. These methods fill an existing string list or list object with the names of tables, stored procedures, fields, or indexes, or with parameter descriptors. This technique is the same as the way you fill lists with metadata for any other database connection component. These methods are described in "Obtaining metadata" on page 23-13.

If you require more detailed schema information, you can populate a unidirectional dataset with metadata. Instead of a simple list, the unidirectional dataset is filled with schema information, where each record represents a single table, stored procedure, index, field, or parameter.

### Fetching metadata into a unidirectional dataset

To populate a unidirectional datasets with metadata from the database server, you must first indicate what data you want to see, using the *SetSchemaInfo* method. *SetSchemaInfo* takes three parameters:

- The type of schema information (metadata) you want to fetch. This can be a list of tables (*stTables*), a list of system tables (*stSysTables*), a list of stored procedures (*stProcedures*), a list of fields in a table (*stColumns*), a list of indexes (*stIndexes*), or a list of parameters used by a stored procedure (*stProcedureParams*). Each type of information uses a different set of fields to describe the items in the list. For details on the structures of these datasets, see "The structure of metadata datasets" on page 28-14.
- If you are fetching information about fields, indexes, or stored procedure parameters, the name of the table or stored procedure to which they apply. If you are fetching any other type of schema information, this parameter is nil.

- A pattern that must be matched for every name returned. This pattern is an SQL pattern such as 'Cust%', which uses the wildcards '%' (to match a string of arbitrary characters of any length) and '_' (to match a single arbitrary character). To use a literal percent or underscore in a pattern, the character is doubled (%% or __). If you do not want to use a pattern, this parameter can be nil.

**Note** If you are fetching schema information about tables (*stTables*), the resulting schema information can describe ordinary tables, system tables, views, and/or synonyms, depending on the value of the SQL connection's *TableScope* property.

The following call requests a table listing all system tables (server tables that contain metadata):

```
SQLDataSet1.SetSchemaInfo(stSysTable, '', '');
```

When you open the dataset after this call to *SetSchemaInfo*, the resulting dataset has a record for each table, with columns giving the table name, type, schema name, and so on. If the server does not use system tables to store metadata (for example MySQL), when you open the dataset it contains no records.

The previous example used only the first parameter. Suppose, Instead, you want to obtain a list of input parameters for a stored procedure named 'MyProc'. Suppose, further, that the person who wrote that stored procedure named all parameters using a prefix to indicate whether they were input or output parameters ('inName', 'outValue' and so on). You could call *SetSchemaInfo* as follows:

```
SQLDataSet1.SetSchemaInfo(stProcedureParams, 'MyProc', 'in%');
```

The resulting dataset is a table of input parameters with columns to describe the properties of each parameter.

## Fetching data after using the dataset for metadata

There are two ways to return to executing queries or stored procedures with the dataset after a call to *SetSchemaInfo*:

- Change the *CommandText* property, specifying the query, table, or stored procedure from which you want to fetch data.
- Call *SetSchemaInfo*, setting the first parameter to *stNoSchema*. In this case, the dataset reverts to fetching the data specified by the current value of *CommandText*.

## The structure of metadata datasets

For each type of metadata you can access using *TSQLDataSet*, there is a predefined set of columns (fields) that are populated with information about the items of the requested type.

### Information about tables

When you request information about tables (*stTables* or *stSysTables*), the resulting dataset includes a record for each table. It has the following columns:

**Table 28.1** Columns in tables of metadata listing tables

| Column name | Field type | Contents |
|---|---|---|
| RECNO | ftInteger | A record number that uniquely identifies each record. |
| CATALOG_NAME | ftString | The name of the catalog (database) that contains the table. This is the same as the *Database* parameter on an SQL connection component. |
| SCHEMA_NAME | ftString | The name of the schema that identifies the owner of the table. |
| TABLE_NAME | ftString | The name of the table. This field determines the sort order of the dataset. |
| TABLE_TYPE | ftInteger | Identifies the type of table. It is a sum of one or more of the following values:<br>1: Table<br>2: View<br>4: System table<br>8: Synonym<br>16: Temporary table<br>32: Local table. |

### Information about stored procedures

When you request information about stored procedures (*stProcedures*), the resulting dataset includes a record for each stored procedure. It has following columns:

**Table 28.2** Columns in tables of metadata listing stored procedures

| Column name | Field type | Contents |
|---|---|---|
| RECNO | ftInteger | A record number that uniquely identifies each record. |
| CATALOG_NAME | ftString | The name of the catalog (database) that contains the stored procedure. This is the same as the *Database* parameter on an SQL connection component. |
| SCHEMA_NAME | ftString | The name of the schema that identifies the owner of the stored procedure. |
| PROC_NAME | ftString | The name of the stored procedure. This field determines the sort order of the dataset. |
| PROC_TYPE | ftInteger | Identifies the type of stored procedure. It is a sum of one or more of the following values:<br>1: Procedure<br>2: Function<br>4: Package<br>8: System procedure |
| IN_PARAMS | ftSmallint | The number of input parameters |
| OUT_PARAMS | ftSmallint | The number of output parameters. |

### Information about fields

When you request information about the fields in a specified table (*stColumns*), the resulting dataset includes a record for each field. It includes the following columns:

**Table 28.3** Columns in tables of metadata listing fields

| Column name | Field type | Contents |
|---|---|---|
| RECNO | ftInteger | A record number that uniquely identifies each record. |
| CATALOG_NAME | ftString | The name of the catalog (database) that contains the table whose fields you listing. This is the same as the *Database* parameter on an SQL connection component. |
| SCHEMA_NAME | ftString | The name of the schema that identifies the owner of the field. |
| TABLE_NAME | ftString | The name of the table that contains the fields. |
| COLUMN_NAME | ftString | The name of the field. This value determines the sort order of the dataset. |
| COLUMN_POSITION | ftSmallint | The position of the column in its table. |
| COLUMN_TYPE | ftInteger | Identifies the type of value in the field. It is a sum of one or more of the following:<br>1: Row ID<br>2: Row Version<br>4: Auto increment field<br>8: Field with a default value |
| COLUMN_DATATYPE | ftSmallint | The datatype of the column. This is one of the logical field type constants defined in sqllinks.pas. |
| COLUMN_TYPENAME | ftString | A string describing the datatype. This is the same information as contained in COLUMN_DATATYPE and COLUMN_SUBTYPE, but in a form used in some DDL statements. |
| COLUMN_SUBTYPE | ftSmallint | A subtype for the column's datatype. This is one of the logical subtype constants defined in sqllinks.pas. |
| COLUMN_PRECISION | ftInteger | The size of the field type (number of characters in a string, bytes in a bytes field, significant digits in a BCD value, members of an ADT field, and so on). |
| COLUMN_SCALE | ftSmallint | The number of digits to the right of the decimal on BCD values, or descendants on ADT and array fields. |
| COLUMN_LENGTH | ftInteger | The number of bytes required to store field values. |
| COLUMN_NULLABLE | ftSmallint | A Boolean that indicates whether the field can be left blank (0 means the field requires a value). |

### Information about indexes

When you request information about the indexes on a table (stIndexes), the resulting dataset includes a record for each field in each record. (Multi-record indexes are described using multiple records) The dataset has the following columns:

**Table 28.4** Columns in tables of metadata listing indexes

| Column name | Field type | Contents |
|---|---|---|
| RECNO | ftInteger | A record number that uniquely identifies each record. |
| CATALOG_NAME | ftString | The name of the catalog (database) that contains the index. This is the same as the *Database* parameter on an SQL connection component. |
| SCHEMA_NAME | ftString | The name of the schema that identifies the owner of the index. |
| TABLE_NAME | ftString | The name of the table for which the index is defined. |
| INDEX_NAME | ftString | The name of the index. This field determines the sort order of the dataset. |
| PKEY_NAME | ftString | Indicates the name of the primary key. |
| COLUMN_NAME | ftString | The name of the field (column) in the index. |
| COLUMN_POSITION | ftSmallint | The position of this field in the index. |
| INDEX_TYPE | ftSmallint | Identifies the type of index. It is a sum of one or more of the following values:<br>1: Non-unique<br>2: Unique<br>4: Primary key |
| SORT_ORDER | ftString | Indicates that the index is ascending (a) or descending (d). |
| FILTER | ftString | Describes a filter condition that limits the indexed records. |

### Information about stored procedure parameters

When you request information about the parameters of a stored procedure (*stProcedureParams*), the resulting dataset includes a record for each parameter. It has the following columns:

**Table 28.5** Columns in tables of metadata listing parameters

| Column name | Field type | Contents |
|---|---|---|
| RECNO | ftInteger | A record number that uniquely identifies each record. |
| CATALOG_NAME | ftString | The name of the catalog (database) that contains the stored procedure. This is the same as the *Database* parameter on an SQL connection component. |
| SCHEMA_NAME | ftString | The name of the schema that identifies the owner of the stored procedure. |
| PROC_NAME | ftString | The name of the stored procedure that contains the parameter. |
| PARAM_NAME | ftString | The name of the parameter. This field determines the sort order of the dataset. |
| PARAM_TYPE | ftSmallint | Identifies the type of parameter. This is the same as a *TParam* object's *ParamType* property. |
| PARAM_DATATYPE | ftSmallint | The datatype of the parameter. This is one of the logical field type constants defined in sqllinks.pas. |
| PARAM_SUBTYPE | ftSmallint | A subtype for the parameter's datatype. This is one of the logical subtype constants defined in sqllinks.pas. |
| PARAM_TYPENAME | ftString | A string describing the datatype. This is the same information as contained in PARAM_DATATYPE and PARAM_SUBTYPE, but in a form used in some DDL statements. |
| PARAM_PRECISION | ftInteger | The maximum number of digits in floating-point values or bytes (for strings and Bytes fields). |
| PARAM_SCALE | ftSmallint | The number of digits to the right of the decimal on floating-point values. |
| PARAM_LENGTH | ftInteger | The number of bytes required to store parameter values. |
| PARAM_NULLABLE | ftSmallint | A Boolean that indicates whether the parameter can be left blank (0 means the parameter requires a value). |

## Debugging dbExpress applications

While you are debugging your database application, it may prove useful to monitor the SQL messages that are sent to and from the database server through your connection component, including those that are generated automatically for you (for example by a provider component or by the *dbExpress* driver).

### Using TSQLMonitor to monitor SQL commands

*TSQLConnection* uses a companion component, *TSQLMonitor*, to intercept these messages and save them in a string list. *TSQLMonitor* works much like the SQL monitor utility that you can use with the BDE, except that it monitors only those commands involving a single *TSQLConnection* component rather than all commands managed by *dbExpress*.

To use *TSQLMonitor*,

1. Add a *TSQLMonitor* component to the form or data module containing the *TSQLConnection* component whose SQL commands you want to monitor.
2. Set its *SQLConnection* property to the *TSQLConnection* component.
3. Set the SQL monitor's *Active* property to *True*.

As SQL commands are sent to the server, the SQL monitor's *TraceList* property is automatically updated to list all the SQL commands that are intercepted.

You can save this list to a file by specifying a value for the *FileName* property and then setting the *AutoSave* property to *True*. *AutoSave* causes the SQL monitor to save the contents of the *TraceList* property to a file every time is logs a new message.

If you do not want the overhead of saving a file every time a message is logged, you can use the *OnLogTrace* event handler to only save files after a number of messages have been logged. For example, the following event handler saves the contents of *TraceList* every 10th message, clearing the log after saving it so that the list never gets too long:

```
procedure TForm1.SQLMonitor1LogTrace(Sender: TObject; CBInfo: Pointer);
var
  LogFileName: string;
begin
  with Sender as TSQLMonitor do
  begin
    if TraceCount = 10 then
    begin
      LogFileName := 'c:\log' + IntToStr(Tag) + '.txt';
      Tag := Tag + 1; {ensure next log file has a different name }
      SaveToFile(LogFileName);
      TraceList.Clear; { clear list }
    end;
  end;
end;
```

**Note** If you were to use the previous event handler, you would also want to save any partial list (fewer than 10 entries) when the application shuts down.

## Using a callback to monitor SQL commands

Instead of using *TSQLMonitor*, you can customize the way your application traces SQL commands by using the SQL connection component's *SetTraceCallbackEvent* method. *SetTraceCallbackEvent* takes two parameters: a callback of type *TSQLCallbackEvent*, and a user-defined value that is passed to the callback function.

The callback function takes two parameters: *CallType* and *CBInfo*:

- *CallType* is reserved for future use.
- *CBInfo* is a pointer to a record that includes the category (the same as *CallType*), the text of the SQL command, and the user-defined value that is passed to the *SetTraceCallbackEvent* method.

The callback returns a value of type *CBRType*, typically *cbrUSEDEF*.

The *dbExpress* driver calls your callback every time the SQL connection component passes a command to the server or the server returns an error message.

**Warning** Do not call *SetTraceCallbackEvent* if the *TSQLConnection* object has an associated *TSQLMonitor* component. *TSQLMonitor* uses the callback mechanism to work, and *TSQLConnection* can only support one callback at a time.

Chapter

29

# Using client datasets

Client datasets are specialized datasets that hold all their data in memory. The support for manipulating the data they store in memory is provided by midaslib.dcu or midas.dll. The format client datasets use for storing data is self-contained and easily transported, which allows client datasets to

- Read from and write to dedicated files on disk, acting as a file-based dataset. Properties and methods supporting this mechanism are described in "Using a client dataset with file-based data" on page 29-33.
- Cache updates for data from a database server. Client dataset features that support cached updates are described in "Using a client dataset to cache updates" on page 29-16.
- Represent the data in the client portion of a multi-tiered application. To function in this way, the client dataset must work with an external provider, as described in "Using a client dataset with a provider" on page 29-24. For information about multi-tiered database applications, see Chapter 31, "Creating multi-tiered applications."
- Represent the data from a source other than a dataset. Because a client dataset can use the data from an external provider, specialized providers can adapt a variety of information sources to work with client datasets. For example, you can use an XML provider to enable a client dataset to represent the information in an XML document.

Whether you use client datasets for file-based data, caching updates, data from an external provider (such as working with an XML document or in a multi-tiered application), or a combination of these approaches such as a "briefcase model" application, you can take advantage of broad range of features client datasets support for working with data.

## Working with data using a client dataset

Like any dataset, you can use client datasets to supply the data for data-aware controls using a data source component. See Chapter 20, "Using data controls"for information on how to display database information in data-aware controls.

Client datasets implement all the properties an methods inherited from *TDataSet*. For a complete introduction to this generic dataset behavior, see Chapter 24, "Understanding datasets."

In addition, client datasets implement many of the features common to table type datasets such as

- Sorting records with indexes.
- Using Indexes to search for records.
- Limiting records with ranges.
- Creating master/detail relationships.
- Controlling read/write access
- Creating the underlying dataset
- Emptying the dataset
- Synchronizing client datasets

For details on these features, see "Using table type datasets" on page 24-25.

Client datasets differ from other datasets in that they hold all their data in memory. Because of this, their support for some database functions can involve additional capabilities or considerations. This chapter describes some of these common functions and the differences introduced by client datasets.

### Navigating data in client datasets

If an application uses standard data-aware controls, then a user can navigate through a client dataset's records using the built-in behavior of those controls. You can also navigate programmatically through records using standard dataset methods such as *First*, *Last*, *Next*, and *Prior*. For more information about these methods, see "Navigating datasets" on page 24-5.

Unlike most datasets, client datasets can also position the cursor at a specific record in the dataset by using the *RecNo* property. Ordinarily an application uses *RecNo* to determine the record number of the current record. Client datasets can, however, set *RecNo* to a particular record number to make that record the current one.

### Limiting what records appear

To restrict users to a subset of available data on a temporary basis, applications can use ranges and filters. When you apply a range or a filter, the client dataset does not display all the data in its in-memory cache. Instead, it only displays the data that meets the range or filter conditions. For more information about using filters, see "Displaying and editing a subset of data using filters" on page 24-13. For more information about ranges, see "Limiting records with ranges" on page 24-31.

With most datasets, filter strings are parsed into SQL commands that are then implemented on the database server. Because of this, the SQL dialect of the server limits what operations are used in filter strings. Client datasets implement their own filter support, which includes more operations than that of other datasets. For example, when using a client dataset, filter expressions can include string operators that return substrings, operators that parse date/time values, and much more. Client datasets also allow filters on BLOB fields or complex field types such as ADT fields and array fields.

The various operators and functions that client datasets can use in filters, along with a comparison to other datasets that support filters, is given below:

**Table 29.1** Filter support in client datasets

| Operator or function | Example | Supported by other datasets | Comment |
|---|---|---|---|
| **Comparisons** | | | |
| = | State = 'CA' | Yes | |
| <> | State <> 'CA' | Yes | |
| >= | DateEntered >= '1/1/1998' | Yes | |
| <= | Total <= 100,000 | Yes | |
| > | Percentile > 50 | Yes | |
| < | Field1 < Field2 | Yes | |
| BLANK | State <> 'CA' or State = BLANK | Yes | Blank records do not appear unless explicitly included in the filter. |
| IS NULL | Field1 IS NULL | No | |
| IS NOT NULL | Field1 IS NOT NULL | No | |
| **Logical operators** | | | |
| and | State = 'CA' and Country = 'US' | Yes | |
| or | State = 'CA' or State = 'MA' | Yes | |
| not | not (State = 'CA') | Yes | |
| **Arithmetic operators** | | | |
| + | Total + 5 > 100 | Depends on driver | Applies to numbers, strings, or date (time) + number. |
| - | Field1 - 7 <> 10 | Depends on driver | Applies to numbers, dates, or date (time) - number. |
| * | Discount * 100 > 20 | Depends on driver | Applies to numbers only. |
| / | Discount > Total / 5 | Depends on driver | Applies to numbers only. |

**Table 29.1** Filter support in client datasets (continued)

| Operator or function | Example | Supported by other datasets | Comment |
|---|---|---|---|
| **String functions** | | | |
| Upper | Upper(Field1) = 'ALWAYS' | No | |
| Lower | Lower(Field1 + Field2) = 'josp' | No | |
| Substring | Substring(DateFld,8) = '1998'<br>Substring(DateFld,1,3) = 'JAN' | No | Value goes from position of second argument to end or number of chars in third argument. First char has position 1. |
| Trim | Trim(Field1 + Field2)<br>Trim(Field1, '-') | No | Removes third argument from front and back. If no third argument, trims spaces. |
| TrimLeft | TrimLeft(StringField)<br>TrimLeft(Field1, '$') <> '' | No | See Trim. |
| TrimRight | TrimRight(StringField)<br>TrimRight(Field1, '.') <> '' | No | See Trim. |
| **DateTime functions** | | | |
| Year | Year(DateField) = 2000 | No | |
| Month | Month(DateField) <> 12 | No | |
| Day | Day(DateField) = 1 | No | |
| Hour | Hour(DateField) < 16 | No | |
| Minute | Minute(DateField) = 0 | No | |
| Second | Second(DateField) = 30 | No | |
| GetDate | GetDate - DateField > 7 | No | Represents current date and time. |
| Date | DateField = Date(GetDate) | No | Returns the date portion of a datetime value. |
| Time | TimeField > Time(GetDate) | No | Returns the time portion of a datetime value. |
| **Miscellaneous** | | | |
| Like | Memo LIKE '%filters%' | No | Works like SQL-92 without the ESC clause. When applied to BLOB fields, FilterOptions determines whether case is considered. |
| In | Day(DateField) in (1,7) | No | Works like SQL-92. Second argument is a list of values all with the same type. |
| * | State = 'M*' | Yes | Wildcard for partial comparisons. |

When applying ranges or filters, the client dataset still stores all of its records in memory. The range or filter merely determines which records are available to controls that navigate or display data from the client dataset.

**Note** When fetching data from a provider, you can also limit the data that the client dataset stores by supplying parameters to the provider. For details, see "Limiting records with parameters" on page 29-29.

## Editing data

Client datasets represent their data as an in-memory data packet. This packet is the value of the client dataset's *Data* property. By default, however, edits are not stored in the *Data* property. Instead the insertions, deletions, and modifications (made by users or programmatically) are stored in an internal change log, represented by the *Delta* property. Using a change log serves two purposes:

- The change log is required for applying updates to a database server or external provider component.
- The change log provides sophisticated support for undoing changes.

The *LogChanges* property lets you disable logging. When *LogChanges* is *True*, changes are recorded in the log. When *LogChanges* is *False*, changes are made directly to the *Data* property. You can disable the change log in file-based applications if you do not want the undo support.

Edits in the change log remain there until they are removed by the application. Applications remove edits when

- Undoing changes
- Saving changes

**Note** Saving the client dataset to a file does not remove edits from the change log. When you reload the dataset, the *Data* and *Delta* properties are the same as they were when the data was saved.

### Undoing changes

Even though a record's original version remains unchanged in *Data*, each time a user edits a record, leaves it, and returns to it, the user sees the last changed version of the record. If a user or application edits a record a number of times, each changed version of the record is stored in the change log as a separate entry.

Storing each change to a record makes it possible to support multiple levels of undo operations should it be necessary to restore a record's previous state:

- To remove the last change to a record, call *UndoLastChange*. *UndoLastChange* takes a Boolean parameter, *FollowChange*, that indicates whether to reposition the cursor on the restored record (*True*), or to leave the cursor on the current record (*False*). If there are several changes to a record, each call to *UndoLastChange* removes another layer of edits. *UndoLastChange* returns a Boolean value indicating success or failure. If the removal occurs, *UndoLastChange* returns *True*. Use the *ChangeCount* property to check whether there are more changes to undo. *ChangeCount* indicates the number of changes stored in the change log.
- Instead of removing each layer of changes to a single record, you can remove them all at once. To remove all changes to a record, select the record, and call *RevertRecord*. *RevertRecord* removes any changes to the current record from the change log.
- To restore a deleted record, first set the *StatusFilter* property to [*usDeleted*], which makes the deleted records "visible." Next, navigate to the record you want to restore and call *RevertRecord*. Finally, restore the *StatusFilter* property to [*usModified*, *usInserted*, *usUnmodified*] so that the edited version of the dataset (now containing the restored record) is again visible.
- At any point during edits, you can save the current state of the change log using the *SavePoint* property. Reading *SavePoint* returns a marker into the current position in the change log. Later, if you want to undo all changes that occurred since you read the save point, set *SavePoint* to the value you read previously. Your application can obtain values for multiple save points. However, once you back up the change log to a save point, the values of all save points that your application read after that one are invalid.
- You can abandon all changes recorded in the change log by calling *CancelUpdates*. *CancelUpdates* clears the change log, effectively discarding all edits to all records. Be careful when you call *CancelUpdates*. After you call *CancelUpdates*, you cannot recover any changes that were in the log.

## Saving changes

Client datasets use different mechanisms for incorporating changes from the change log, depending on whether the client datasets stores its data in a file or represents data obtained through a provider. Whichever mechanism is used, the change log is automatically emptied when all updates have been incorporated.

File-based applications can simply merge the changes into the local cache represented by the *Data* property. They do not need to worry about resolving local edits with changes made by other users. To merge the change log into the *Data* property, call the *MergeChangeLog* method. "Merging changes into data" on page 29-34 describes this process.

You can't use *MergeChangeLog* if you are using the client dataset to cache updates or to represent the data from an external provider component. The information in the change log is required for resolving updated records with the data stored in the database (or source dataset). Instead, you call *ApplyUpdates*, which attempts to write the modifications to the database server or source dataset, and updates the *Data* property only when the modifications have been successfully committed. See "Applying updates" on page 29-20 for more information about this process.

## Constraining data values

Client datasets can enforce constraints on the edits a user makes to data. These constraints are applied when the user tries to post changes to the change log. You can always supply custom constraints. These let you provide your own, application-defined limits on what values users post to a client dataset.

In addition, when client datasets represent server data that is accessed using the BDE, they also enforce data constraints imported from the database server. If the client dataset works with an external provider component, the provider can control whether those constraints are sent to the client dataset, and the client dataset can control whether it uses them. For details on how the provider controls whether constraints are included in data packets, see "Handling server constraints" on page 30-13. For details on how and why client dataset can turn off enforcement of server constraints, see "Handling constraints from the server" on page 29-30.

### Specifying custom constraints

You can use the properties of the client dataset's field components to impose your own constraints on what data users can enter. Each field component has two properties that you can use to specify constraints:

- The *DefaultExpression* property defines a default value that is assigned to the field if the user does not enter a value. Note that if the database server or source dataset also assigns a default expression for the field, the client dataset's version takes precedence because it is assigned before the update is applied back to the database server or source dataset.
- The *CustomConstraint* property lets you assign a constraint condition that must be met before a field value can be posted. Custom constraints defined this way are applied in addition to any constraints imported from the server. For more information about working with custom constraints on field components, see "Creating a custom constraint" on page 25-22.

In addition, you can create record-level constraints using the client dataset's *Constraints* property. *Constraints* is a collection of *TCheckConstraint* objects, where each object represents a separate condition. Use the *CustomConstraint* property of a *TCheckConstraint* object to add your own constraints that are checked when you post records.

## Sorting and indexing

Using indexes provides several benefits to your applications:

- They allow client datasets to locate data quickly.
- They let you apply ranges to limit the available records.
- They let your application set up relationships with other datasets such as lookup tables or master/detail forms.
- They specify the order in which records appear.

If a client dataset represents server data or uses an external provider, it inherits a default index and sort order based on the data it receives. The default index is called DEFAULT_ORDER. You can use this ordering, but you cannot change or delete the index.

In addition to the default index, the client dataset maintains a second index, called CHANGEINDEX, on the changed records stored in the change log (*Delta* property). CHANGEINDEX orders all records in the client dataset as they would appear if the changes specified in *Delta* were applied. CHANGEINDEX is based on the ordering inherited from DEFAULT_ORDER. As with DEFAULT_ORDER, you cannot change or delete the CHANGEINDEX index.

You can use other existing indexes, and you can create your own indexes. The following sections describe how to create and use indexes with client datasets.

**Note** You may also want to review the material on indexes in table type datasets, which also applies to client datasets. This material is in "Sorting records with indexes" on page 24-26 and "Limiting records with ranges" on page 24-31.

### Adding a new index

There are three ways to add indexes to a client dataset:

- To create a temporary index at runtime that sorts the records in the client dataset, you can use the *IndexFieldNames* property. Specify field names, separated by semicolons. Ordering of field names in the list determines their order in the index.

  This is the least powerful method of adding indexes. You can't specify a descending or case-insensitive index, and the resulting indexes do not support grouping. These indexes do not persist when you close the dataset, and are not saved when you save the client dataset to a file.

- To create an index at runtime that can be used for grouping, call *AddIndex*. *AddIndex* lets you specify the properties of the index, including
  - The name of the index. This can be used for switching indexes at runtime.
  - The fields that make up the index. The index uses these fields to sort records and to locate records that have specific values on these fields.

  - How the index sorts records. By default, indexes impose an ascending sort order (based on the machine's locale). This default sort order is case-sensitive. You can set options to make the entire index case-insensitive or to sort in descending order. Alternately, you can provide a list of fields to be sorted case-insensitively and a list of fields to be sorted in descending order.
  - The default level of grouping support for the index.

  Indexes created with *AddIndex* do not persist when the client dataset is closed. (That is, they are lost when you reopen the client dataset). You can't call *AddIndex* when the dataset is closed. Indexes you add using *AddIndex* are not saved when you save the client dataset to a file.

- The third way to create an index is at the time the client dataset is created. Before creating the client dataset, specify the desired indexes using the *IndexDefs* property. The indexes are then created along with the underlying dataset when you call *CreateDataSet*. See "Creating and deleting tables" on page 24-38 for more information about creating client datasets.

  As with *AddIndex*, indexes you create with the dataset support grouping, can sort in ascending order on some fields and descending order on others, and can be case insensitive on some fields and case sensitive on others. Indexes created this way always persist and are saved when you save the client dataset to a file.

**Tip** You can index and sort on internally calculated fields with client datasets.

### Deleting and switching indexes

To remove an index you created for a client dataset, call *DeleteIndex* and specify the name of the index to remove. You cannot remove the DEFAULT_ORDER and CHANGEINDEX indexes.

To use a different index when more than one index is available, use the *IndexName* property to select the index to use. At design time, you can select from available indexes in *IndexName* property drop-down box in the Object Inspector.

### Using indexes to group data

When you use an index in your client dataset, it automatically imposes a sort order on the records. Because of this order, adjacent records usually contain duplicate values on the fields that make up the index. For example, consider the following fragment from an orders table that is indexed on the SalesRep and Customer fields:

| SalesRep | Customer | OrderNo | Amount |
|---|---|---|---|
| 1 | 1 | 5 | 100 |
| 1 | 1 | 2 | 50 |
| 1 | 2 | 3 | 200 |
| 1 | 2 | 6 | 75 |
| 2 | 1 | 1 | 10 |
| 2 | 3 | 4 | 200 |

Because of the sort order, adjacent values in the SalesRep column are duplicated. Within the records for SalesRep 1, adjacent values in the Customer column are duplicated. That is, the data is grouped by SalesRep, and within the SalesRep group it is grouped by Customer. Each grouping has an associated level. In this case, the SalesRep group has level 1 (because it is not nested in any other groups) and the Customer group has level 2 (because it is nested in the group with level 1). Grouping level corresponds to the order of fields in the index.

Client datasets let you determine where the current record lies within any given grouping level. This allows your application to display records differently, depending on whether they are the first record in the group, in the middle of a group, or the last record in a group. For example, you might want to display a field value only if it is on the first record of the group, eliminating the duplicate values. To do this with the previous table results in the following:

| SalesRep | Customer | OrderNo | Amount |
|---|---|---|---|
| 1 | 1 | 5 | 100 |
| | | 2 | 50 |
| | 2 | 3 | 200 |
| | | 6 | 75 |
| 2 | 1 | 1 | 10 |
| | 3 | 4 | 200 |

To determine where the current record falls within any group, use the *GetGroupState* method. *GetGroupState* takes an integer giving the level of the group and returns a value indicating where the current record falls the group (first record, last record, or neither).

When you create an index, you can specify the level of grouping it supports (up to the number of fields in the index). *GetGroupState* can't provide information about groups beyond that level, even if the index sorts records on additional fields.

## Representing calculated values

As with any dataset, you can add calculated fields to your client dataset. These are fields whose values you calculate dynamically, usually based on the values of other fields in the same record. For more information about using calculated fields, see "Defining a calculated field" on page 25-7.

Client datasets, however, let you optimize when fields are calculated by using internally calculated fields. For more information on internally calculated fields, see "Using internally calculated fields in client datasets" below.

You can also tell client datasets to create calculated values that summarize the data in several records using maintained aggregates. For more information on maintained aggregates, see "Using maintained aggregates" on page 29-11.

### Using internally calculated fields in client datasets

In other datasets, your application must compute the value of calculated fields every time the record changes or the user edits any fields in the current record. It does this in an *OnCalcFields* event handler.

While you can still do this, client datasets let you minimize the number of times calculated fields must be recomputed by saving calculated values in the client dataset's data. When calculated values are saved with the client dataset, they must still be recomputed when the user edits the current record, but your application need not recompute values every time the current record changes. To save calculated values in the client dataset's data, use internally calculated fields instead of calculated fields.

Internally calculated fields, just like calculated fields, are calculated in an *OnCalcFields* event handler. However, you can optimize your event handler by checking the *State* property of your client dataset. When *State* is *dsInternalCalc*, you must recompute internally calculated fields. When *State* is *dsCalcFields*, you need only recompute regular calculated fields.

To use internally calculated fields, you must define the fields as internally calculated before you create the client dataset. Depending on whether you use persistent fields or field definitions, you do this in one of the following ways:

- If you use persistent fields, define fields as internally calculated by selecting InternalCalc in the Fields editor.
- If you use field definitions, set the *InternalCalcField* property of the relevant field definition to *True*.

**Note** Other types of datasets use internally calculated fields. However, with other datasets, you do not calculate these values in an *OnCalcFields* event handler. Instead, they are computed automatically by the BDE or remote database server.

## Using maintained aggregates

Client datasets provide support for summarizing data over groups of records. Because these summaries are automatically updated as you edit the data in the dataset, this summarized data is called a "maintained aggregate."

In their simplest form, maintained aggregates let you obtain information such as the sum of all values in a column of the client dataset. They are flexible enough, however, to support a variety of summary calculations and to provide subtotals over groups of records defined by a field in an index that supports grouping.

### Specifying aggregates

To specify that you want to calculate summaries over the records in a client dataset, use the *Aggregates* property. *Aggregates* is a collection of aggregate specifications (*TAggregate*). You can add aggregate specifications to your client dataset using the Collection Editor at design time, or using the *Add* method of *Aggregates* at runtime. If you want to create field components for the aggregates, create persistent fields for the aggregated values in the Fields Editor.

**Note** When you create aggregated fields, the appropriate aggregate objects are added to the client dataset's *Aggregates* property automatically. Do not add them explicitly when creating aggregated persistent fields. For details on creating aggregated persistent fields, see "Defining an aggregate field" on page 25-10.

For each aggregate, the *Expression* property indicates the summary calculation it represents. *Expression* can contain a simple summary expression such as

```
Sum(Field1)
```

or a complex expression that combines information from several fields, such as

```
Sum(Qty * Price) - Sum(AmountPaid)
```

Aggregate expressions include one or more of the summary operators in Table 29.2

**Table 29.2** Summary operators for maintained aggregates

| Operator | Use |
|---|---|
| Sum | Totals the values for a numeric field or expression |
| Avg | Computes the average value for a numeric or date-time field or expression |
| Count | Specifies the number of non-blank values for a field or expression |
| Min | Indicates the minimum value for a string, numeric, or date-time field or expression |
| Max | Indicates the maximum value for a string, numeric, or date-time field or expression |

The summary operators act on field values or on expressions built from field values using the same operators you use to create filters. (You can't nest summary operators, however.) You can create expressions by using operators on summarized values with other summarized values, or on summarized values and constants. However, you can't combine summarized values with field values, because such expressions are ambiguous (there is no indication of which record should supply the field value.) These rules are illustrated in the following expressions:

```
Sum(Qty * Price)            {legal -- summary of an expression on fields }
Max(Field1) - Max(Field2)   {legal -- expression on summaries }
Avg(DiscountRate) * 100     {legal -- expression of summary and constant }
Min(Sum(Field1))            {illegal -- nested summaries }
Count(Field1) - Field2      {illegal -- expression of summary and field }
```

## Aggregating over groups of records

By default, maintained aggregates are calculated so that they summarize all the records in the client dataset. However, you can specify that you want to summarize over the records in a group instead. This lets you provide intermediate summaries such as subtotals for groups of records that share a common field value.

Before you can specify a maintained aggregate over a group of records, you must use an index that supports the appropriate grouping. See "Using indexes to group data" on page 29-9 for information on grouping support.

Once you have an index that groups the data in the way you want it summarized, specify the *IndexName* and *GroupingLevel* properties of the aggregate to indicate what index it uses, and which group or subgroup on that index defines the records it summarizes.

For example, consider the following fragment from an orders table that is grouped by SalesRep and, within SalesRep, by Customer:

| SalesRep | Customer | OrderNo | Amount |
|---|---|---|---|
| 1 | 1 | 5 | 100 |
| 1 | 1 | 2 | 50 |
| 1 | 2 | 3 | 200 |
| 1 | 2 | 6 | 75 |
| 2 | 1 | 1 | 10 |
| 2 | 3 | 4 | 200 |

The following code sets up a maintained aggregate that indicates the total amount for each sales representative:

```
Agg.Expression := 'Sum(Amount)';
Agg.IndexName := 'SalesCust';
Agg.GroupingLevel := 1;
Agg.AggregateName := 'Total for Rep';
```

To add an aggregate that summarizes for each customer within a given sales representative, create a maintained aggregate with level 2.

Maintained aggregates that summarize over a group of records are associated with a specific index. The *Aggregates* property can include aggregates that use different indexes. However, only the aggregates that summarize over the entire dataset and those that use the current index are valid. Changing the current index changes which aggregates are valid. To determine which aggregates are valid at any time, use the *ActiveAggs* property.

### Obtaining aggregate values

To get the value of a maintained aggregate, call the *Value* method of the *TAggregate* object that represents the aggregate. *Value* returns the maintained aggregate for the group that contains the current record of the client dataset.

When you are summarizing over the entire client dataset, you can call *Value* at any time to obtain the maintained aggregate. However, when you are summarizing over grouped information, you must be careful to ensure that the current record is in the group whose summary you want. Because of this, it is a good idea to obtain aggregate values at clearly specified times, such as when you move to the first record of a group or when you move to the last record of a group. Use the *GetGroupState* method to determine where the current record falls within a group.

To display maintained aggregates in data-aware controls, use the Fields editor to create a persistent aggregate field component. When you specify an aggregate field in the Fields editor, the client dataset's *Aggregates* is automatically updated to include the appropriate aggregate specification. The *AggFields* property contains the new aggregated field component, and the *FindField* method returns it.

## Copying data from another dataset

To copy the data from another dataset at design time, right click the client dataset and choose Assign Local Data. A dialog appears listing all the datasets available in your project. Select the one whose data and structure you want to copy and choose OK. When you copy the source dataset, your client dataset is automatically activated.

To copy from another dataset at runtime, you can assign its data directly or, if the source is another client dataset, you can clone the cursor.

### Assigning data directly

You can use the client dataset's *Data* property to assign data to a client dataset from another dataset. *Data* is a data packet in the form of an OleVariant. A data packet can come from another client dataset or from any other dataset by using a provider. Once a data packet is assigned to *Data*, its contents are displayed automatically in data-aware controls connected to the client dataset by a data source component.

When you open a client dataset that represents server data or that uses an external provider component, data packets are automatically assigned to *Data*.

When your client dataset does not use a provider, you can copy the data from another client dataset as follows:

```
ClientDataSet1.Data := ClientDataSet2.Data;
```

**Note** When you copy the *Data* property of another client dataset, you copy the change log as well, but the copy does not reflect any filters or ranges that have been applied. To include filters or ranges, you must clone the source dataset's cursor instead.

If you are copying from a dataset other than a client dataset, you can create a dataset provider component, link it to the source dataset, and then copy its data:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SourceDataSet;
ClientDataSet1.Data := TempProvider.Data;
TempProvider.Free;
```

**Note** When you assign directly to the *Data* property, the new data packet is not merged into the existing data. Instead, all previous data is replaced.

If you want to merge changes from another dataset, rather than copying its data, you must use a provider component. Create a dataset provider as in the previous example, but attach it to the destination dataset and instead of copying the data property, use the *ApplyUpdates* method:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := ClientDataSet1;
TempProvider.ApplyUpdates(SourceDataSet.Delta, -1, ErrCount);
TempProvider.Free;
```

### Cloning a client dataset cursor

Client datasets use the *CloneCursor* method to let you work with a second view of the data at runtime. *CloneCursor* lets a second client dataset share the original client dataset's data. This is less expensive than copying all the original data, but, because the data is shared, the second client dataset can't modify the data without affecting the original client dataset.

*CloneCursor* takes three parameters: *Source* specifies the client dataset to clone. The last two parameters (*Reset* and *KeepSettings*) indicate whether to copy information other than the data. This information includes any filters, the current index, links to a master table (when the source dataset is a detail set), the *ReadOnly* property, and any links to a connection component or provider.

When *Reset* and *KeepSettings* are *False*, a cloned client dataset is opened, and the settings of the source client dataset are used to set the properties of the destination. When *Reset* is *True*, the destination dataset's properties are given the default values (no index or filters, no master table, *ReadOnly* is *False*, and no connection component or provider is specified). When *KeepSettings* is *True*, the destination dataset's properties are not changed.

## Adding application-specific information to the data

Application developers can add custom information to the client dataset's *Data* property. Because this information is bundled with the data packet, it is included when you save the data to a file or stream. It is copied when you copy the data to another dataset. Optionally, it can be included with the *Delta* property so that a provider can read this information when it receives updates from the client dataset.

To save application-specific information with the *Data* property, use the *SetOptionalParam* method. This method lets you store an OleVariant that contains the data under a specific name.

To retrieve this application-specific information, use the *GetOptionalParam* method, passing in the name that was used when the information was stored.

## Using a client dataset to cache updates

By default, when you edit data in most datasets, every time you delete or post a record, the dataset generates a transaction, deletes or writes that record to the database server, and commits the transaction. If there is a problem writing changes to the database, your application is notified immediately: the dataset raises an exception when you post the record.

If your dataset uses a remote database server, this approach can degrade performance due to network traffic between your application and the server every time you move to a new record after editing the current record. To minimize the network traffic, you may want to cache updates locally. When you cache updates, you application retrieves data from the database, caches and edits it locally, and then applies the cached updates to the database in a single transaction. When you cache updates, changes to a dataset (such as posting changes or deleting records) are stored locally instead of being written directly to the dataset's underlying table. When changes are complete, your application calls a method that writes the cached changes to the database and clears the cache.

Caching updates can minimize transaction times and reduce network traffic. However, cached data is local to your application and is not under transaction control. This means that while you are working on your local, in-memory, copy of the data, other applications can be changing the data in the underlying database table. They also can't see any changes you make until you apply the cached updates. Because of this, cached updates may not be appropriate for applications that work with volatile data, as you may create or encounter too many conflicts when trying to merge your changes into the database.

Although the BDE and ADO provide alternate mechanisms for caching updates, using a client dataset for caching updates has several advantages:

- Applying updates when datasets are linked in master/detail relationships is handled for you. This ensures that updates to multiple linked datasets are applied in the correct order.
- Client datasets give you the maximum of control over the update process. You can set properties to influence the SQL that is generated for updating records, specify the table to use when updating records from a multi-table join, or even apply updates manually from a *BeforeUpdateRecord* event handler.
- When errors occur applying cached updates to the database server, only client datasets (and dataset providers) provide you with information about the current record value on the database server in addition to the original (unedited) value from your dataset and the new (edited) value of the update that failed.
- Client datasets let you specify the number of update errors you want to tolerate before the entire update is rolled back.

## Overview of using cached updates

To use cached updates, the following order of processes must occur in an application:

1 **Indicate the data you want to edit.** How you do this depends on the type of client dataset you are using:

- If you are using *TClientDataSet*, Specify the provider component that represent the data you want to edit. This is described in “Specifying a provider” on page 29-25.
- If you are using a client dataset associated with a particular data access mechanism, you must
  - Identify the database server by setting the *DBConnection* property to an appropriate connection component.
  - Indicate what data you want to see by specifying the *CommandText* and *CommandType* properties. *CommandType* indicates whether *CommandText* is an SQL statement to execute, the name of a stored procedure, or the name of a table. If *CommandText* is a query or stored procedure, use the *Params* property to provide any input parameters.
  - Optionally, use the *Options* property to indicate whether nested detail sets and BLOB data should be included in data packets or fetched separately, whether specific types of edits (insertions, modifications, or deletions) should be disabled, whether a single update can affect multiple server records, and whether the client dataset’s records are refreshed when it applies updates. *Options* is identical to a provider’s *Options* property. As a result, it allows you to set options that are not relevant or appropriate. For example, there is no reason to include *poIncFieldProps*, because the client dataset does not fetch its data from a dataset with persistent fields. Conversely, you do not want to exclude *poAllowCommandText*, which is included by default, because that would disable the *CommandText* property, which the client dataset uses to specify what data it wants. For information on the provider’s *Options* property, see “Setting options that influence the data packets” on page 30-5.

2 **Display and edit the data,** permit insertion of new records, and support deletions of existing records. Both the original copy of each record and any edits to it are stored in memory. This process is described in “Editing data” on page 29-5.

3 **Fetch additional records as necessary.** By default, client datasets fetch all records and store them in memory. If a dataset contains many records or records with large BLOB fields, you may want to change this so that the client dataset fetches only enough records for display and re-fetches as needed. For details on how to control the record-fetching process, see “Requesting data from the source dataset or document” on page 29-26.

4 **Optionally, refresh the records.** As time passes, other users may modify the data on the database server. This can cause the client dataset’s data to deviate more and more from the data on the server, increasing the chance of errors when you apply updates. To mitigate this problem, you can refresh records that have not already been edited. See “Refreshing records” on page 29-31 for details.

5 **Apply the locally cached records to the database** or cancel the updates. For each record written to the database, a *BeforeUpdateRecord* event is triggered. If an error occurs when writing an individual record to the database, an *OnUpdateError* event enables the application to correct the error, if possible, and continue updating. When updates are complete, all successfully applied updates are cleared from the local cache. For more information about applying updates to the database, see "Updating records" on page 29-20.

   Instead of applying updates, an application can cancel the updates, emptying the change log without writing the changes to the database. You can cancel the updates by calling *CancelUpdates* method. All deleted records in the cache are undeleted, modified records revert to original values, and newly inserted record simply disappear.

## Choosing the type of dataset for caching updates

Delphi includes some specialized client dataset components for caching updates. Each client dataset is associated with a particular data access mechanism. These are listed in Table 29.3:

**Table 29.3** Specialized client datasets for caching updates

| Client dataset | Data access mechanism |
|---|---|
| TBDEClientDataSet | Borland Database Engine |
| TSimpleDataSet | dbExpress |
| TIBClientDataSet | InterBase Express |

In addition, you can cache updates using the generic client dataset (*TClientDataSet*) with an external provider and source dataset. For information about using *TClientDataSet* with an external provider, see "Using a client dataset with a provider" on page 29-24.

**Note** The specialized client datasets associated with each data access mechanism actually use a provider and source dataset as well. However, both the provider and the source dataset are internal to the client dataset.

It is simplest to use one of the specialized client datasets to cache updates. However, there are times when it is preferable to use *TClientDataSet* with an external provider:

- If you are using a data access mechanism that does not have a specialized client dataset, you must use *TClientDataSet* with an external provider component. For example, if the data comes from an XML document or custom dataset.
- If you are working with tables that are related in a master/detail relationship, you should use *TClientDataSet* and connect it, using a provider, to the master table of two source datasets linked in a master/detail relationship. The client dataset sees the detail dataset as a nested dataset field. This approach is necessary so that updates to master and detail tables can be applied in the correct order.

- If you want to code event handlers that respond to the communication between the client dataset and the provider (for example, before and after the client dataset fetches records from the provider), you must use *TClientDataSet* with an external provider component. The specialized client datasets publish the most important events for applying updates (*OnReconcileError*, *BeforeUpdateRecord* and *OnGetTableName*), but do not publish the events surrounding communication between the client dataset and its provider, because they are intended primarily for multi-tiered applications.
- When using the BDE, you may want to use an external provider and source dataset if you need to use an update object. Although it is possible to code an update object from the *BeforeUpdateRecord* event handler of *TBDEClientDataSet*, it can be simpler just to assign the *UpdateObject* property of the source dataset. For information about using update objects, see "Using update objects to update a dataset" on page 26-40.

## Indicating what records are modified

While the user edits a client dataset, you may find it useful to provide feedback about the edits that have been made. This is especially useful if you want to allow the user to undo specific edits, for example, by navigating to them and clicking an "Undo" button.

The *UpdateStatus* method and *StatusFilter* properties are useful when providing feedback on what updates have occurred:

- *UpdateStatus* indicates what type of update, if any, has occurred for the current record. It can be any of the following values:
  - *usUnmodified* indicates that the current record is unchanged.
  - *usModified* indicates that the current record has been edited.
  - *usInserted* indicates a record that was inserted by the user.
  - *usDeleted* indicates a record that was deleted by the user.
- *StatusFilter* controls what type of updates in the change log are visible. *StatusFilter* works on cached records in much the same way as filters work on regular data. *StatusFilter* is a set, so it can contain any combination of the following values:
  - *usUnmodified* indicates an unmodified record.
  - *usModified* indicates a modified record.
  - *usInserted* indicates an inserted record.
  - *usDeleted* indicates a deleted record.

  By default, *StatusFilter* is the set [*usModified*, *usInserted*, *usUnmodified*]. You can add *usDeleted* to this set to provide feedback about deleted records as well.

**Note** *UpdateStatus* and *StatusFilter* are also useful in *BeforeUpdateRecord* and *OnReconcileError* event handlers. For information about *BeforeUpdateRecord*, see "Intervening as updates are applied" on page 29-21. For information about *OnReconcileError*, see "Reconciling update errors" on page 29-23.

The following example shows how to provide feedback about the update status of records using the *UpdateStatus* method. It assumes that you have changed the *StatusFilter* property to include *usDeleted*, allowing deleted records to remain visible in the dataset. It further assumes that you have added a calculated field to the dataset called “Status.”

```
procedure TForm1.ClientDataSet1CalcFields(DataSet: TDataSet);
begin
  with ClientDataSet1 do begin
    case UpdateStatus of
      usUnmodified: FieldByName('Status').AsString := '';
      usModified: FieldByName('Status').AsString := 'M';
      usInserted: FieldByName('Status').AsString := 'I';
      usDeleted: FieldByName('Status').AsString := 'D';
    end;
  end;
end;
```

## Updating records

The contents of the change log are stored as a data packet in the client dataset’s *Delta* property. To make the changes in *Delta* permanent, the client dataset must apply them to the database (or source dataset or XML document).

When a client applies updates to the server, the following steps occur:

1. The client application calls the *ApplyUpdates* method of a client dataset object. This method passes the contents of the client dataset’s *Delta* property to the (internal or external) provider. *Delta* is a data packet that contains a client dataset’s updated, inserted, and deleted records.
2. The provider applies the updates, caching any problem records that it can’t resolve itself. See “Responding to client update requests” on page 30-8 for details on how the provider applies updates.
3. The provider returns all unresolved records to the client dataset in a *Result* data packet. The *Result* data packet contains all records that were not updated. It also contains error information, such as error messages and error codes.
4. The client dataset attempts to reconcile update errors returned in the *Result* data packet on a record-by-record basis.

### Applying updates

Changes made to the client dataset’s local copy of data are not sent to the database server (or XML document) until the client application calls the *ApplyUpdates* method. *ApplyUpdates* takes the changes in the change log, and sends them as a data packet (called *Delta*) to the provider. (Note that, when using most client datasets, the provider is internal to the client dataset.)

*ApplyUpdates* takes a single parameter, *MaxErrors*, which indicates the maximum number of errors that the provider should tolerate before aborting the update process. If *MaxErrors* is *0*, then as soon as an update error occurs, the entire update process is terminated. No changes are written to the database, and the client dataset's change log remains intact. If *MaxErrors* is *-1*, any number of errors is tolerated, and the change log contains all records that could not be successfully applied. If *MaxErrors* is a positive value, and more errors occur than are permitted by *MaxErrors*, all updates are aborted. If fewer errors occur than specified by *MaxErrors*, all records successfully applied are automatically cleared from the client dataset's change log.

*ApplyUpdates* returns the number of actual errors encountered, which is always less than or equal to *MaxErrors* plus one. This return value indicates the number of records that could not be written to the database.

The client dataset's *ApplyUpdates* method does the following:

1. It indirectly calls the provider's *ApplyUpdates* method. The provider's *ApplyUpdates* method writes the updates to the database, source dataset, or XML document and attempts to correct any errors it encounters. Records that it cannot apply because of error conditions are sent back to the client dataset.
2. The client dataset 's *ApplyUpdates* method then attempts to reconcile these problem records by calling the *Reconcile* method. *Reconcile* is an error-handling routine that calls the *OnReconcileError* event handler. You must code the *OnReconcileError* event handler to correct errors. For details about using *OnReconcileError*, see "Reconciling update errors" on page 29-23.
3. Finally, *Reconcile* removes successfully applied changes from the change log and updates *Data* to reflect the newly updated records. When *Reconcile* completes, *ApplyUpdates* reports the number of errors that occurred.

**Important** In some cases, the provider can't determine how to apply updates (for example, when applying updates from a stored procedure or multi-table join). Client datasets and provider components generate events that let you handle these situations. See "Intervening as updates are applied" below for details.

**Tip** If the provider is on a stateless application server, you may want to communicate with it about persistent state information before or after you apply updates. *TClientDataSet* receives a *BeforeApplyUpdates* event before the updates are sent, which lets you send persistent state information to the server. After the updates are applied (but before the reconcile process), *TClientDataSet* receives an *AfterApplyUpdates* event where you can respond to any persistent state information returned by the application server.

## Intervening as updates are applied

When a client dataset applies its updates, the provider determines how to handle writing the insertions, deletions, and modifications to the database server or source dataset. When you use *TClientDataSet* with an external provider component, you can use the properties and events of that provider to influence the way updates are applied. These are described in "Responding to client update requests" on page 30-8.

When the provider is internal, however, as it is for any client dataset associated with a data access mechanism, you can't set its properties or provide event handlers. As a result, the client dataset publishes one property and two events that let you influence how the internal provider applies updates.

- *UpdateMode* controls what fields are used to locate records in the SQL statements the provider generates for applying updates. *UpdateMode* is identical to the provider's *UpdateMode* property. For information on the provider's *UpdateMode* property, see "Influencing how updates are applied" on page 30-10.
- *OnGetTableName* lets you supply the provider with the name of the database table to which it should apply updates. This lets the provider generate the SQL statements for updates when it can't identify the database table from the stored procedure or query specified by *CommandText*. For example, if the query executes a multi-table join that only requires updates to a single table, supplying an *OnGetTableName* event handler allows the internal provider to correctly apply updates.

  An *OnGetTableName* event handler has three parameters: the internal provider component, the internal dataset that fetched the data from the server, and a parameter to return the table name to use in the generated SQL.
- *BeforeUpdateRecord* occurs for every record in the delta packet. This event lets you make any last-minute changes before the record is inserted, deleted, or modified. It also provides a way for you to execute your own SQL statements to apply the update in cases where the provider can't generate correct SQL (for example, for multi-table joins where multiple tables must be updated.)

  A *BeforeUpdateRecord* event handler has five parameters: the internal provider component, the internal dataset that fetched the data from the server, a delta packet that is positioned on the record that is about to be updated, an indication of whether the update is an insertion, deletion, or modification, and a parameter that returns whether the event handler performed the update.The use of these is illustrated in the following event handler. For simplicity, the example assumes the SQL statements are available as global variables that only need field values:

```
procedure TForm1.SimpleDataSet1BeforeUpdateRecord(Sender: TObject;
   SourceDS: TDataSet; DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind;
   var Applied Boolean);
var
  SQL: string;
  Connection: TSQLConnection;
begin
  Connection := (SourceDS as TSimpleDataSet).Connection;
  case UpdateKind of
  ukModify:
    begin
     { 1st dataset: update Fields[1], use Fields[0] in where clause }
      SQL := Format(UpdateStmt1, [DeltaDS.Fields[1].NewValue, DeltaDS.Fields[0].OldValue]);
      Connection.Execute(SQL, nil, nil);
     { 2nd dataset: update Fields[2], use Fields[3] in where clause }
      SQL := Format(UpdateStmt2, [DeltaDS.Fields[2].NewValue, DeltaDS.Fields[3].OldValue]);
      Connection.Execute(SQL, nil, nil);
    end;
  ukDelete:
    begin
     { 1st dataset: use Fields[0] in where clause }
      SQL := Format(DeleteStmt1, [DeltaDS.Fields[0].OldValue]);
      Connection.Execute(SQL, nil, nil);
     { 2nd dataset: use Fields[3] in where clause }
      SQL := Format(DeleteStmt2, [DeltaDS.Fields[3].OldValue]);
      Connection.Execute(SQL, nil, nil);
    end;
  ukInsert:
    begin
     { 1st dataset: values in Fields[0] and Fields[1] }
       SQL := Format(InsertStmt1, [DeltaDS.Fields[0].NewValue, DeltaDS.Fields[1].NewValue]);
      Connection.Execute(SQL, nil, nil);
     { 2nd dataset: values in Fields[2] and Fields[3] }
      SQL := Format(InsertStmt2, [DeltaDS.Fields[2].NewValue, DeltaDS.Fields[3].NewValue]);
      Connection.Execute(SQL, nil, nil);
    end;
  end;
  Applied := True;
end;
```

## Reconciling update errors

There are two events that let you handle errors that occur during the update process:

- During the update process, the internal provider generates an *OnUpdateError* event every time it encounters an update that it can't handle. If you correct the problem in an *OnUpdateError* event handler, then the error does not count toward the maximum number of errors passed to the *ApplyUpdates* method. This event only occurs for client datasets that use an internal provider. If you are using *TClientDataSet*, you can use the provider component's *OnUpdateError* event instead.
- After the entire update operation is finished, the client dataset generates an *OnReconcileError* event for every record that the provider could not apply to the database server.

You should always code an *OnReconcileError* or *OnUpdateError* event handler, even if only to discard the records returned that could not be applied. The event handlers for these two events work the same way. They include the following parameters:

- *DataSet:* A client dataset that contains the updated record which couldn't be applied. You can use this dataset's methods to get information about the problem record and to edit the record in order to correct any problems. In particular, you will want to use the *CurValue*, *OldValue*, and *NewValue* properties of the fields in the current record to determine the cause of the update problem. However, you must not call any client dataset methods that change the current record in your event handler.
- *E:* An object that represents the problem that occurred. You can use this exception to extract an error message or to determine the cause of the update error.

- *UpdateKind:* The type of update that generated the error. *UpdateKind* can be *ukModify* (the problem occurred updating an existing record that was modified), *ukInsert* (the problem occurred inserting a new record), or *ukDelete* (the problem occurred deleting an existing record).
- *Action:* A **var** parameter that indicates what action to take when the event handler exits. In your event handler, you set this parameter to
  - Skip this record, leaving it in the change log. (rrSkip or raSkip)
  - Stop the entire reconcile operation. (rrAbort or raAbort)
  - Merge the modification that failed into the corresponding record from the server. (rrMerge or raMerge) This only works if the server record does not include any changes to fields modified in the client dataset's record.
  - Replace the current update in the change log with the value of the record in the event handler, which has presumably been corrected. (rrApply or raCorrect)
  - Ignore the error completely. (rrIgnore) This possibility only exists in the *OnUpdateError* event handler, and is intended for the case where the event handler applies the update back to the database server. The updated record is removed from the change log and merged into *Data*, as if the provider had applied the update.
  - Back out the changes for this record on the client dataset, reverting to the originally provided values. (raCancel) This possibility only exists in the *OnReconcileError* event handler.
  - Update the current record value to match the record on the server. (raRefresh) This possibility only exists in the *OnReconcileError* event handler.

The following code shows an *OnReconcileError* event handler that uses the reconcile error dialog from the RecError unit which ships in the objrepos directory. (To use this dialog, add RecError to your uses clause.)

```
procedure TForm1.ClientDataSetReconcileError(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind, var Action TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

## Using a client dataset with a provider

A client dataset uses a provider to supply it with data and apply updates when

- It caches updates from a database server or another dataset.
- It represents the data in an XML document.
- It stores the data in the client portion of a multi-tiered application.

For any client dataset other than *TClientDataSet*, this provider is internal, and so not directly accessible by the application. With *TClientDataSet*, the provider is an external component that links the client dataset to an external source of data.

An external provider component can reside in the same application as the client dataset, or it can be part of a separate application running on another system. For more information about provider components, see Chapter 30, "Using provider components." For more information about applications where the provider is in a separate application on another system, see Chapter 31, "Creating multi-tiered applications."

When using an (internal or external) provider, the client dataset always caches any updates. For information on how this works, see "Using a client dataset to cache updates" on page 29-16.

The following topics describe additional properties and methods of the client dataset that enable it to work with a provider.

## Specifying a provider

Unlike the client datasets that are associated with a data access mechanism, *TClientDataSet* has no internal provider component to package data or apply updates. If you want it to represent data from a source dataset or XML document, therefore, you must associated the client dataset with an external provider component.

The way you associate *TClientDataSet* with a provider depends on whether the provider is in the same application as the client dataset or on a remote application server running on another system.

- If the provider is in the same application as the client dataset, you can associate it with a provider by choosing a provider from the drop-down list for the *ProviderName* property in the Object Inspector. This works as long as the provider has the same *Owner* as the client dataset. (The client dataset and the provider have the same *Owner* if they are placed in the same form or data module.) To use a local provider that has a different *Owner*, you must form the association at runtime using the client dataset's *SetProvider* method

  If you think you may eventually scale up to a remote provider, or if you want to make calls directly to the *IAppServer* interface, you can also set the *RemoteServer* property to a *TLocalConnection* component. If you use *TLocalConnection*, the *TLocalConnection* instance manages the list of all providers that are local to the application, and handles the client dataset's *IAppServer* calls. If you do not use *TLocalConnection*, the application creates a hidden object that handles the *IAppServer* calls from the client dataset.

- If the provider is on a remote application server, then, in addition to the *ProviderName* property, you need to specify a component that connects the client dataset to the application server. There are two properties that can handle this task: *RemoteServer*, which specifies the name of a connection component from which to get a list of providers, or *ConnectionBroker*, which specifies a centralized broker that provides an additional level of indirection between the client dataset and the connection component. The connection component and, if used, the connection broker, reside in the same data module as the client dataset. The connection component establishes and maintains a connection to an application server, sometimes called a "data broker". For more information, see "The structure of the client application" on page 31-4.

  At design time, after you specify *RemoteServer* or *ConnectionBroker*, you can select a provider from the drop-down list for the *ProviderName* property in the Object Inspector. This list includes both local providers (in the same form or data module) and remote providers that can be accessed through the connection component.

**Note** If the connection component is an instance of *TDCOMConnection*, the application server must be registered on the client machine.

At runtime, you can switch among available providers (both local and remote) by setting *ProviderName* in code.

## Requesting data from the source dataset or document

Client datasets can control how they fetch their data packets from a provider. By default, they retrieve all records from the source dataset. This is true whether the source dataset and provider are internal components (as with *TBDEClientDataSet, TSimpleDataSet,* and *TIBClientDataSet*), or separate components that supply the data for *TClientDataSet*.

You can change how the client dataset fetches records using the *PacketRecords* and *FetchOnDemand* properties.

### Incremental fetching

By changing the *PacketRecords* property, you can specify that the client dataset fetches data in smaller chunks. *PacketRecords* specifies either how many records to fetch at a time, or the type of records to return. By default, *PacketRecords* is set to *-1*, which means that all available records are fetched at once, either when the client dataset is first opened, or when the application explicitly calls *GetNextPacket*. When *PacketRecords* is *-1*, then after the client dataset first fetches data, it never needs to fetch more data because it already has all available records.

To fetch records in small batches, set *PacketRecords* to the number of records to fetch. For example, the following statement sets the size of each data packet to ten records:

```
ClientDataSet1.PacketRecords := 10;
```

This process of fetching records in batches is called "incremental fetching". Client datasets use incremental fetching when *PacketRecords* is greater than zero.

To fetch each batch of records, the client dataset calls *GetNextPacket*. Newly fetched packets are appended to the end of the data already in the client dataset. *GetNextPacket* returns the number of records it fetches. If the return value is the same as *PacketRecords*, the end of available records was not encountered. If the return value is greater than *0* but less than *PacketRecords*, the last record was reached during the fetch operation. If *GetNextPacket* returns *0*, then there are no more records to fetch.

**Warning** Incremental fetching does not work if you are fetching data from a remote provider on a stateless application server. See "Supporting state information in remote data modules" on page 31-19 for information on how to use incremental fetching with stateless remote data modules.

**Note** You can also use *PacketRecords* to fetch metadata information about the source dataset. To retrieve metadata information, set *PacketRecords* to *0*.

### Fetch-on-demand

Automatic fetching of records is controlled by the *FetchOnDemand* property. When *FetchOnDemand* is *True* (the default), the client dataset automatically fetches records as needed. To prevent automatic fetching of records, set *FetchOnDemand* to *False*. When *FetchOnDemand* is *False*, the application must explicitly call *GetNextPacket* to fetch records.

For example, Applications that need to represent extremely large read-only datasets can turn off *FetchOnDemand* to ensure that the client datasets do not try to load more data than can fit into memory. Between fetches, the client dataset frees its cache using the *EmptyDataSet* method. This approach, however, does not work well when the client must post updates to the server.

The provider controls whether the records in data packets include BLOB data and nested detail datasets. If the provider excludes this information from records, the *FetchOnDemand* property causes the client dataset to automatically fetch BLOB data and detail datasets on an as-needed basis. If *FetchOnDemand* is *False*, and the provider does not include BLOB data and detail datasets with records, you must explicitly call the *FetchBlobs* or *FetchDetails* method to retrieve this information.

## Getting parameters from the source dataset

There are two circumstances when the client dataset needs to fetch parameter values:

- The application needs the value of output parameters on a stored procedure.
- The application wants to initialize the input parameters of a query or stored procedure to the current values on the source dataset.

Client datasets store parameter values in their *Params* property. These values are refreshed with any output parameters when the client dataset fetches data from the source dataset. However, there may be times a *TClientDataSet* component in a client application needs output parameters when it is not fetching data.

To fetch output parameters when not fetching records, or to initialize input parameters, the client dataset can request parameter values from the source dataset by calling the *FetchParams* method. The parameters are returned in a data packet from the provider and assigned to the client dataset's *Params* property.

At design time, the *Params* property can be initialized by right-clicking the client dataset and choosing Fetch Params.

**Note** There is never a need to call *FetchParams* when the client dataset uses an internal provider and source dataset, because the *Params* property always reflects the parameters of the internal source dataset. With *TClientDataSet*, the *FetchParams* method (or the Fetch Params command) only works if the client dataset is connected to a provider whose associated dataset can supply parameters. For example, if the source dataset is a table type dataset, there are no parameters to fetch.

If the provider is on a separate system as part of a stateless application server, you can't use *FetchParams* to retrieve output parameters. In a stateless application server, other clients can change and rerun the query or stored procedure, changing output parameters before the call to *FetchParams*. To retrieve output parameters from a stateless application server, use the *Execute* method. If the provider is associated with a query or stored procedure, *Execute* tells the provider to execute the query or stored procedure and return any output parameters. These returned parameters are then used to automatically update the *Params* property.

## Passing parameters to the source dataset

Client datasets can pass parameters to the source dataset to specify what data they want provided in the data packets it sends. These parameters can specify

- Input parameter values for a query or stored procedure that is run on the application server
- Field values that limit the records sent in data packets

You can specify parameter values that your client dataset sends to the source dataset at design time or at runtime. At design time, select the client dataset and double-click the *Params* property in the Object Inspector. This brings up the collection editor, where you can add, delete, or rearrange parameters. By selecting a parameter in the collection editor, you can use the Object Inspector to edit the properties of that parameter.

At runtime, use the *CreateParam* method of the *Params* property to add parameters to your client dataset. *CreateParam* returns a parameter object, given a specified name, parameter type, and datatype. You can then use the properties of that parameter object to assign a value to the parameter.

For example, the following code adds an input parameter named CustNo with a value of 605:

```
with ClientDataSet1.Params.CreateParam(ftInteger, 'CustNo', ptInput) do
  AsInteger := 605;
```

If the client dataset is not active, you can send the parameters to the application server and retrieve a data packet that reflects those parameter values simply by setting the *Active* property to *True*.

### Sending query or stored procedure parameters

When the client dataset's *CommandType* property is *ctQuery* or *ctStoredProc*, or, if the client dataset is a *TClientDataSet* instance, when the associated provider represents the results of a query or stored procedure, you can use the *Params* property to specify parameter values. When the client dataset requests data from the source dataset or uses its *Execute* method to run a query or stored procedure that does not return a dataset, it passes these parameter values along with the request for data or the execute command. When the provider receives these parameter values, it assigns them to its associated dataset. It then instructs the dataset to execute its query or stored procedure using these parameter values, and, if the client dataset requested data, begins providing data, starting with the first record in the result set.

**Note** Parameter names should match the names of the corresponding parameters on the source dataset.

### Limiting records with parameters

If the client dataset is

- a *TClientDataSet* instance whose associated provider represents a *TTable* or *TSQLTable* component
- a *TSimpleDataSet* or a *TBDEClientDataSet* instance whose *CommandType* property is *ctTable*

then it can use the *Params* property to limit the records that it caches in memory. Each parameter represents a field value that must be matched before a record can be included in the client dataset's data. This works much like a filter, except that with a filter, the records are still cached in memory, but unavailable.

Each parameter name must match the name of a field. When using *TClientDataSet*, these are the names of fields in the *TTable* or *TSQLTable* component associated with the provider. When using *TSimpleDataSet* or *TBDEClientDataSet*, these are the names of fields in the table on the database server. The data in the client dataset then includes only those records whose values on the corresponding fields match the values assigned to the parameters.

For example, consider an application that displays the orders for a single customer. When the user identifies the customer, the client dataset sets its *Params* property to include a single parameter named CustID (or whatever field in the source table is called) whose value identifies the customer whose orders should be displayed. When the client dataset requests data from the source dataset, it passes this parameter value. The provider then sends only the records for the identified customer. This is more efficient than letting the provider send all the orders records to the client application and then filtering the records using the client dataset.

## Handling constraints from the server

When a database server defines constraints on what data is valid, it is useful if the client dataset knows about them. That way, the client dataset can ensure that user edits never violate those server constraints. As a result, such violations are never passed to the database server where they would be rejected. This means fewer updates generate error conditions during the updating process.

Regardless of the source of data, you can duplicate such server constraints by explicitly adding them to the client dataset. This process is described in “Specifying custom constraints” on page 29-7.

It is more convenient, however, if the server constraints are automatically included in data packets. Then you need not explicitly specify default expressions and constraints, and the client dataset changes the values it enforces when the server constraints change. By default, this is exactly what happens: if the source dataset is aware of server constraints, the provider automatically includes them in data packets and the client dataset enforces them when the user posts edits to the change log.

**Note** Only datasets that use the BDE can import constraints from the server. This means that server constraints are only included in data packets when using *TBDEClientDataSet* or *TClientDataSet* with a provider that represents a BDE-based dataset. For more information on how to import server constraints and how to prevent a provider from including them in data packets, see “Handling server constraints” on page 30-13.

**Note** For more information on working with the constraints once they have been imported, see “Using server constraints” on page 25-23.

While importing server constraints and expressions is an extremely valuable feature that helps an application preserve data integrity, there may be times when it needs to disable constraints on a temporary basis. For example, if a server constraint is based on the current maximum value of a field, but the client dataset uses incremental fetching, the current maximum value for a field in the client dataset may differ from the maximum value on the database server, and constraints may be invoked differently. In another case, if a client dataset applies a filter to records when constraints are enabled, the filter may interfere in unintended ways with constraint conditions. In each of these cases, an application may disable constraint-checking.

To disable constraints temporarily, call the *DisableConstraints* method. Each time *DisableConstraints* is called, a reference count is incremented. While the reference count is greater than zero, constraints are not enforced on the client dataset.

To reenable constraints for the client dataset, call the dataset’s *EnableConstraints* method. Each call to *EnableConstraints* decrements the reference count. When the reference count is zero, constraints are enabled again.

**Tip** Always call *DisableConstraints* and *EnableConstraints* in paired blocks to ensure that constraints are enabled when you intend them to be.

## Refreshing records

Client datasets work with an in-memory snapshot of the data from the source dataset. If the source dataset represents server data, then as time elapses other users may modify that data. The data in the client dataset becomes a less accurate picture of the underlying data.

Like any other dataset, client datasets have a *Refresh* method that updates its records to match the current values on the server. However, calling *Refresh* only works if there are no edits in the change log. Calling *Refresh* when there are unapplied edits results in an exception.

Client datasets can also update the data while leaving the change log intact. To do this, call the *RefreshRecord* method. Unlike the *Refresh* method, *RefreshRecord* updates only the current record in the client dataset. *RefreshRecord* changes the record value originally obtained from the provider but leaves any changes in the change log.

**Warning** It is not always appropriate to call *RefreshRecord*. If the user's edits conflict with changes made to the underlying dataset by other users, calling *RefreshRecord* masks this conflict. When the client dataset applies its updates, no reconcile error occurs and the application can't resolve the conflict.

In order to avoid masking update errors, you may want to check that there are no pending updates before calling *RefreshRecord*. For example, the following *AfterScroll* refreshes the current record every time the user moves to a new record (ensuring the most up-to-date value), but only when it is safe to do so.:

```
procedure TForm1.ClientDataSet1AfterScroll(DataSet: TDataSet);
begin
  if ClientDataSet1.UpdateStatus = usUnModified then
    ClientDataSet1.RefreshRecord;
end;
```

## Communicating with providers using custom events

Client datasets communicate with a provider component through a special interface called *IAppServer*. If the provider is local, *IAppServer* is the interface to an automatically-generated object that handles all communication between the client dataset and its provider. If the provider is remote, *IAppServer* is the interface to a remote data module on the application server, or (in the case of a SOAP server) an interface generated by the connection component.

*TClientDataSet* provides many opportunities for customizing the communication that uses the *IAppServer* interface. Before and after every *IAppServer* method call that is directed at the client dataset's provider, *TClientDataSet* receives special events that allow it to communicate arbitrary information with its provider. These events are matched with similar events on the provider. Thus for example, when the client dataset calls its *ApplyUpdates* method, the following events occur:

1 The client dataset receives a *BeforeApplyUpdates* event, where it specifies arbitrary custom information in an OleVariant called *OwnerData*.
2 The provider receives a *BeforeApplyUpdates* event, where it can respond to the *OwnerData* from the client dataset and update the value of *OwnerData* to new information.
3 The provider goes through its normal process of assembling a data packet (including all the accompanying events).
4 The provider receives an *AfterApplyUpdates* event, where it can respond to the current value of *OwnerData* and update it to a value for the client dataset.
5 The client dataset receives an *AfterApplyUpdates* event, where it can respond to the returned value of *OwnerData*.

Every other *IAppServer* method call is accompanied by a similar set of *BeforeXXX* and *AfterXXX* events that let you customize the communication between client dataset and provider.

In addition, the client dataset has a special method, *DataRequest*, whose only purpose is to allow application-specific communication with the provider. When the client dataset calls *DataRequest*, it passes an OleVariant as a parameter that can contain any information you want. This, in turn, generates an is the *OnDataRequest* event on the provider, where you can respond in any application-defined way and return a value to the client dataset.

## Overriding the source dataset

The client datasets that are associated with a particular data access mechanism use the *CommandText* and *CommandType* properties to specify the data they represent. When using *TClientDataSet*, however, the data is specified by the source dataset, not the client dataset. Typically, this source dataset has a property that specifies an SQL statement to generate the data or the name of a database table or stored procedure.

If the provider allows, *TClientDataSet* can override the property on the source dataset that indicates what data it represents. That is, if the provider permits, the client dataset's *CommandText* property replaces the property on the provider's dataset that specifies what data it represents. This allows *TClientDataSet* to specify dynamically what data it wants to see.

By default, external provider components do not let client datasets use the *CommandText* value in this way. To allow *TClientDataSet* to use its *CommandText* property, you must add *poAllowCommandText* to the *Options* property of the provider. Otherwise, the value of *CommandText* is ignored.

**Note** Never remove *poAllowCommandText* from the *Options* property of *TBDEClientDataSet* or *TIBClientDataSet*. The client dataset's *Options* property is forwarded to the internal provider, so removing *poAllowCommandText* prevents the client dataset from specifying what data to access.

The client dataset sends its *CommandText* string to the provider at two times:

- When the client dataset first opens. After it has retrieved the first data packet from the provider, the client dataset does not send *CommandText* when fetching subsequent data packets.
- When the client dataset sends an *Execute* command to provider.

To send an SQL command or to change a table or stored procedure name at any other time, you must explicitly use the *IAppServer* interface that is available as the *AppServer* property. This property represents the interface through which the client dataset communicates with its provider.

## Using a client dataset with file-based data

Client datasets can work with dedicated files on disk as well as server data. This allows them to be used in file-based database applications and "briefcase model" applications. The special files that client datasets use for their data are called MyBase.

**Tip** All client datasets are appropriate for a briefcase model application, but for a pure MyBase application (one that does not use a provider), it is preferable to use *TClientDataSet*, because it involves less overhead.

In a pure MyBase application, the client application cannot get table definitions and data from the server, and there is no server to which it can apply updates. Instead, the client dataset must independently

- Define and create tables
- Load saved data
- Merge edits into its data
- Save data

### Creating a new dataset

There are three ways to define and create client datasets that do not represent server data:

- You can define and create a new client dataset using persistent fields or field and index definitions. This follows the same scheme as creating any table type dataset. See "Creating and deleting tables" on page 24-38 for details.
- You can copy an existing dataset (at design or runtime). See "Copying data from another dataset" on page 29-14 for more information about copying existing datasets.
- You can create a client dataset from an arbitrary XML document. See "Converting XML documents into data packets" on page 32-6 for details.

Once the dataset is created, you can save it to a file. From then on, you do not need to recreate the table, only load it from the file you saved. When beginning a file-based database application, you may want to first create and save empty files for your datasets before writing the application itself. This way, you start with the metadata for your client dataset already defined, making it easier to set up the user interface.

## Loading data from a file or stream

To load data from a file, call a client dataset's *LoadFromFile* method. *LoadFromFile* takes one parameter, a string that specifies the file from which to read data. The file name can be a fully qualified path name, if appropriate. If you always load the client dataset's data from the same file, you can use the *FileName* property instead. If *FileName* names an existing file, the data is automatically loaded when the client dataset is opened.

To load data from a stream, call the client dataset's *LoadFromStream* method. *LoadFromStream* takes one parameter, a stream object that supplies the data.

The data loaded by *LoadFromFile* (*LoadFromStream)* must have previously been saved in a client dataset's data format by this or another client dataset using the *SaveToFile (SaveToStream)* method, or generated from an XML document. For more information about saving data to a file or stream, see "Saving data to a file or stream" on page 29-35. For information about creating client dataset data from an XML document, see Chapter 32, "Using XML in database applications."

When you call *LoadFromFile* or *LoadFromStream*, all data in the file is read into the *Data* property. Any edits that were in the change log when the data was saved are read into the *Delta* property. However, the only indexes that are read from the file are those that were created with the dataset.

## Merging changes into data

When you edit the data in a client dataset, all edits to the data exist only in an in-memory change log. This log can be maintained separately from the data itself, although it is completely transparent to objects that use the client dataset. That is, controls that navigate the client dataset or display its data see a view of the data that includes the changes. If you do not want to back out of changes, however, you should merge the change log into the data of the client dataset by calling the *MergeChangeLog* method. *MergeChangeLog* overwrites records in *Data* with any changed field values in the change log.

After *MergeChangeLog* executes, *Data* contains a mix of existing data and any changes that were in the change log. This mix becomes the new *Data* baseline against which further changes can be made. *MergeChangeLog* clears the change log of all records and resets the *ChangeCount* property to *0*.

**Warning** Do not call *MergeChangeLog* for client datasets that use a provider. In this case, call *ApplyUpdates* to write changes to the database. For more information, see "Applying updates" on page 29-20.

**Note** It is also possible to merge changes into the data of a separate client dataset if that dataset originally provided the data in the *Data* property. To do this, you must use a dataset provider. For an example of how to do this, see "Assigning data directly" on page 29-14.

If you do not want to use the extended undo capabilities of the change log, you can set the client dataset's *LogChanges* property to *False*. When *LogChanges* is *False*, edits are automatically merged when you post records and there is no need to call *MergeChangeLog*.

### Saving data to a file or stream

Even when you have merged changes into the data of a client dataset, this data still exists only in memory. While it persists if you close the client dataset and reopen it in your application, it will disappear when your application shuts down. To make the data permanent, it must be written to disk. Write changes to disk using the *SaveToFile* method.

*SaveToFile* takes one parameter, a string that specifies the file into which to write data. The file name can be a fully qualified path name, if appropriate. If the file already exists, its current contents are completely overwritten.

**Note** *SaveToFile* does not preserve any indexes you added to the client dataset at runtime, only indexes that were added when you created the client dataset.

If you always save the data to the same file, you can use the *FileName* property directly instead. If *FileName* is set, the data is automatically saved to the named file when the client dataset is closed.

You can also save data to a stream, using the *SaveToStream* method. *SaveToStream* takes one parameter, a stream object that receives the data.

**Note** If you save a client dataset while there are still edits in the change log, these are not merged with the data. When you reload the data, using the *LoadFromFile* or *LoadFromStream* method, the change log will still contain the unmerged edits. This is important for applications that support the briefcase model, where those changes will eventually have to be applied to a provider component on the application server.

## Using a simple dataset

*TSimpleDataSet* is a special type of client dataset designed for simple two-tiered applications. Like a unidirectional dataset, it can use an SQL connection component to connect to a database server and specify an SQL statement to execute on that server. Like other client datasets, it buffers data in memory to allow full navigation and editing support.

*TSimpleDataSet* works the same way as a generic client dataset (*TClientDataSet*) that is linked to a unidirectional dataset by a dataset provider. In fact, *TSimpleDataSet* has its own, internal provider, which it uses to communicate with an internally created unidirectional dataset.

Using a simple dataset can simplify the process of two-tiered application development because you don't need to work with as many components.

## When to use TSimpleDataSet

*TSimpleDataSet* is intended for use in a simple two-tiered database applications and briefcase model applications. It provides an easy-to-set up component for linking to the database server, fetching data, caching updates, and applying them back to the server. It can be used in most two-tiered applications.

There are times, however, when it is more appropriate to use *TClientDataSet*:

- If you are not using data from a database server (for example, if you are using a dedicated file on disk), then *TClientDataSet* has the advantage of less overhead.
- Only *TClientDataSet* can be used in a multi-tiered database application. Thus, if you are writing a multi-tiered application, or if you intend to scale up to a multi-tiered application eventually, you should use *TClientDataSet* with an external provider and source dataset.
- Because the source dataset is internal to the simple dataset component, you can't link two source datasets in a master/detail relationship to obtain nested detail sets. (You can, however, link two simple datasets into a master/detail relationship.)
- The simple dataset does not surface any of the events or properties that occur on its internal dataset provider. However, in most cases, these events are used in multi-tiered applications, and are not needed for two-tiered applications.

## Setting up a simple dataset

Setting up a simple dataset requires two essential steps. Set up:

1. The connection information.
2. The dataset information.

The following steps describe setting up a simple dataset in more detail.

To use *TSimpleDataSet*:

1. Place the *TSimpleDataSet* component in a data module or on a form. Set its *Name* property to a unique value appropriate to your application.
2. Identify the database server that contains the data. There are two ways to do this:
   - If you have a named connection in the connections file, expand the *Connection* property and specify the *ConnectionName* value.
   - For greater control over connection properties, transaction support, login support, and the ability to use a single connection for more than one dataset, use a separate *TSQLConnection* component instead. Specify the *TSQLConnection* component as the value of the *Connection* property. For details on *TSQLConnection*, see Chapter 23, "Connecting to databases".

3 To indicate what data you want to fetch from the server, expand the *DataSet* property and set the appropriate values. There are three ways to fetch data from the server:

   - Set *CommandType* to *ctQuery* and set *CommandText* to an SQL statement you want to execute on the server. This statement is typically a SELECT statement. Supply the values for any parameters using the *Params* property.
   - Set *CommandType* to *ctStoredProc* and set *CommandText* to the name of the stored procedure you want to execute. Supply the values for any input parameters using the *Params* property.
   - Set *CommandType* to *ctTable* and set *CommandText* to the name of the database tables whose records you want to use.

4 If the data is to be used with visual data controls, add a data source component to the form or data module, and set its *DataSet* property to the *TSimpleDataSet* object. The data source component forwards the data in the client dataset's in-memory cache to data-aware components for display. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.

5 Activate the dataset by setting the *Active* property to *true* (or, at runtime, calling the *Open* method).

6 If you executed a stored procedure, use the *Params* property to retrieve any output parameters.

7 When the user has edited the data in the simple dataset, you can apply those edits back to the database server by calling the *ApplyUpdates* method. Resolve any update errors in an *OnReconcileError* event handler. For more information on applying updates, see "Updating records" on page 29-20.

Chapter

30

# Using provider components

Provider components (*TDataSetProvider* and *TXMLTransformProvider*) supply the most common mechanism by which client datasets obtain their data. Providers

- Receive data requests from a client dataset (or XML broker), fetch the requested data, package the data into a transportable data packet, and return the data to the client dataset (or XML broker). This activity is called "providing."
- Receive updated data from a client dataset (or XML broker), apply updates to the database server, source dataset, or source XML document, and log any updates that cannot be applied, returning unresolved updates to the client dataset for further reconciliation. This activity is called "resolving."

Most of the work of a provider component happens automatically. You need not write any code on the provider to create data packets from the data in a dataset or XML document or to apply updates. However, provider components include a number of events and properties that allow your application more direct control over what information is packaged for clients and how your application responds to client requests.

When using *TBDEClientDataSet*, *TSimpleDataSet*, or *TIBClientDataSet*, the provider is internal to the client dataset, and the application has no direct access to it. When using *TClientDataSet* or *TXMLBroker*, however, the provider is a separate component that you can use to control what information is packaged for clients and for responding to events that occur around the process of providing and resolving. The client datasets that have internal providers surface some of the internal provider's properties and events as their own properties and events, but for the greatest amount of control, you may want to use *TClientDataSet* with a separate provider component.

When using a separate provider component, it can reside in the same application as the client dataset (or XML broker), or it can reside on an application server as part of a multi-tiered application.

This chapter describes how to use a provider component to control the interaction with client datasets or XML brokers.

## Determining the source of data

When you use a provider component, you must specify the source it uses to get the data it assembles into data packets. Depending on your version of Delphi, you can specify the source as one of the following:

- To provide the data from a dataset, use *TDataSetProvider*.
- To provide the data from an XML document, use *TXMLTransformProvider*.

### Using a dataset as the source of the data

If the provider is a dataset provider (*TDataSetProvider*), set the *DataSet* property of the provider to indicate the source dataset. At design time, select from available datasets in the *DataSet* property drop-down list in the Object Inspector.

*TDataSetProvider* interacts with the source dataset using the *IProviderSupport* interface. This interface is introduced by *TDataSet*, so it is available for all datasets. However, the *IProviderSupport* methods implemented in *TDataSet* are mostly stubs that don't do anything or that raise exceptions.

The dataset classes that ship with Delphi (BDE-enabled datasets, ADO-enabled datasets, *dbExpress* datasets, and InterBase Express datasets) override these methods to implement the *IProviderSupport* interface in a more useful fashion. Client datasets don't add anything to the inherited *IProviderSupport* implementation, but can still be used as a source dataset as long as the *ResolveToDataSet* property of the provider is *True*.

Component writers that create their own custom descendants from *TDataSet* must override all appropriate *IProviderSupport* methods if their datasets are to supply data to a provider. If the provider only provides data packets on a read-only basis (that is, if it does not apply updates), the *IProviderSupport* methods implemented in *TDataSet* may be sufficient.

### Using an XML document as the source of the data

If the provider is an XML provider, set the *XMLDataFile* property of the provider to indicate the source document.

XML providers must transform the source document into data packets, so in addition to indicating the source document, you must also specify how to transform that document into data packets. This transformation is handled by the provider's *TransformRead* property. *TransformRead* represents a *TXMLTransform* object. You can set its properties to specify what transformation to use, and use its events to provide your own input to the transformation. For more information on using XML providers, see "Using an XML document as the source for a provider" on page 32-8.

## Communicating with the client dataset

All communication between a provider and a client dataset or XML broker takes place through an *IAppServer* interface. If the provider is in the same application as the client, this interface is implemented by a hidden object generated automatically for you, or by a *TLocalConnection* component. If the provider is part of a multi-tiered application, this is the interface for the application server's remote data module or (in the case of a SOAP server) an interface generated by the connection component.

Most applications do not use *IAppServer* directly, but invoke it indirectly through the properties and methods of the client dataset or XML broker. However, when necessary, you can make direct calls to the *IAppServer* interface by using the *AppServer* property of a client dataset.

Table 30.1 lists the methods of the *IAppServer* interface, as well as the corresponding methods and events on the provider component and the client dataset. These *IAppServer* methods include a *Provider* parameter. In multi-tiered applications, this parameter indicates the provider on the application server with which the client dataset communicates. Most methods also include an OleVariant parameter called *OwnerData* that allows a client dataset and a provider to pass custom information back and forth. *OwnerData* is not used by default, but is passed to all event handlers so that you can write code that allows your provider to adjust to application-defined information before and after each call from a client dataset.

**Table 30.1** AppServer interface members

| IAppServer | Provider component | TClientDataSet |
|---|---|---|
| AS_ApplyUpdates method | ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event | ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event. |
| AS_DataRequest method | DataRequest method, OnDataRequest event | DataRequest method. |
| AS_Execute method | Execute method, BeforeExecute event, AfterExecute event | Execute method, BeforeExecute event, AfterExecute event. |
| AS_GetParams method | GetParams method, BeforeGetParams event, AfterGetParams event | FetchParams method, BeforeGetParams event, AfterGetParams event. |
| AS_GetProviderNames method | Used to identify all available providers. | Used to create a design-time list for ProviderName property. |
| AS_GetRecords method | GetRecords method, BeforeGetRecords event, AfterGetRecords event | GetNextPacket method, Data property, BeforeGetRecords event, AfterGetRecords event |
| AS_RowRequest method | RowRequest method, BeforeRowRequest event, AfterRowRequest event | FetchBlobs method, FetchDetails method, RefreshRecord method, BeforeRowRequest event, AfterRowRequest event |

## Choosing how to apply updates using a dataset provider

*TXMLTransformProvider* components always apply updates to the associated XML document. When using *TDataSetProvider*, however, you can choose how updates are applied. By default, when *TDataSetProvider* components apply updates and resolve update errors, they communicate directly with the database server using dynamically generated SQL statements. This approach has the advantage that your server application does not need to merge updates twice (first to the dataset, and then to the remote server).

However, you may not always want to take this approach. For example, you may want to use some of the events on the dataset component. Alternately, the dataset you use may not support the use of SQL statements (for example if you are providing from a *TClientDataSet* component).

*TDataSetProvider* lets you decide whether to apply updates to the database server using SQL or to the source dataset by setting the *ResolveToDataSet* property. When this property is *True*, updates are applied to the dataset. When it is *False*, updates are applied directly to the underlying database server.

## Controlling what information is included in data packets

When working with a dataset provider, there are a number of ways to control what information is included in data packets that are sent to and from the client. These include

- Specifying what fields appear in data packets
- Setting options that influence the data packets
- Adding custom information to data packets

**Note** These techniques for controlling the content of data packets are only available for dataset providers. When using *TXMLTransformProvider*, you can only control the content of data packets by controlling the transformation file the provider uses.

### Specifying what fields appear in data packets

When using a dataset provider, you can control what fields are included in data packets by creating persistent fields on the dataset that the provider uses to build data packets. The provider then includes only these fields. Fields whose values are generated dynamically by the source dataset (such as calculated fields or lookup fields) can be included, but appear to client datasets on the receiving end as static read-only fields. For information about persistent fields, see "Persistent field components" on page 25-3.

If the client dataset will be editing the data and applying updates, you must include enough fields so that there are no duplicate records in the data packet. Otherwise, when the updates are applied, it is impossible to determine which record to update. If you do not want the client dataset to be able to see or use extra fields provided only to ensure uniqueness, set the *ProviderFlags* property for those fields to include *pfHidden*.

**Note** Including enough fields to avoid duplicate records is also a consideration when the provider's source dataset represents a query. You must specify the query so that it includes enough fields to ensure all records are unique, even if your application does not use all the fields.

## Setting options that influence the data packets

The *Options* property of a dataset provider lets you specify when BLOBs or nested detail tables are sent, whether field display properties are included, what type of updates are allowed, and so on. The following table lists the possible values that can be included in *Options*.

**Table 30.2** Provider options

| Value | Meaning |
|---|---|
| poAutoRefresh | The provider refreshes the client dataset with current record values whenever it applies updates. |
| poReadOnly | The client dataset can't apply updates to the provider. |
| poDisableEdits | Client datasets can't modify existing data values. If the user tries to edit a field, the client dataset raises exception. (This does not affect the client dataset's ability to insert or delete records). |
| poDisableInserts | Client datasets can't insert new records. If the user tries to insert a new record, the client dataset raises an exception. (This does not affect the client dataset's ability to delete records or modify existing data) |
| poDisableDeletes | Client datasets can't delete records. If the user tries to delete a record, the client dataset raises an exception. (This does not affect the client dataset's ability to insert or modify records) |
| poFetchBlobsOnDemand | BLOB field values are not included in data packets. Instead, client datasets must request these values on an as-needed basis. If the client dataset's *FetchOnDemand* property is *True*, it requests these values automatically. Otherwise, the application must call the client dataset's *FetchBlobs* method to retrieve BLOB data. |
| poFetchDetailsOnDemand | When the provider's dataset represents the master of a master/detail relationship, nested detail values are not included in data packets. Instead, client datasets request these on an as-needed basis. If the client dataset's *FetchOnDemand* property is *True*, it requests these values automatically. Otherwise, the application must call the client dataset's *FetchDetails* method to retrieve nested details. |

**Table 30.2** Provider options (continued)

| Value | Meaning |
|---|---|
| poIncFieldProps | The data packet includes the following field properties (where applicable): *Alignment*, *DisplayLabel*, *DisplayWidth*, *Visible*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, *Currency*, *EditMask*, *DisplayValues.* |
| poCascadeDeletes | When the provider's dataset represents the master of a master/ detail relationship, the server automatically deletes detail records when master records are deleted. To use this option, the database server must be set up to perform cascaded deletes as part of its referential integrity. |
| poCascadeUpdates | When the provider's dataset represents the master of a master/ detail relationship, key values on detail tables are updated automatically when the corresponding values are changed in master records. To use this option, the database server must be set up to perform cascaded updates as part of its referential integrity. |
| poAllowMultiRecordUpdates | A single update can cause more than one record of the underlying database table to change. This can be the result of triggers, referential integrity, SQL statements on the source dataset, and so on. Note that if an error occurs, the event handlers provide access to the record that was updated, not the other records that change in consequence. |
| poNoReset | Client datasets can't specify that the provider should reposition the cursor on the first record before providing data. |
| poPropogateChanges | Changes made by the server to updated records as part of the update process are sent back to the client and merged into the client dataset. |
| poAllowCommandText | The client can override the associated dataset's SQL text or the name of the table or stored procedure it represents. |
| poRetainServerOrder | The client dataset should not re-sort the records in the dataset to enforce a default order. |

## Adding custom information to data packets

Dataset providers can add application-defined information to data packets using the *OnGetDataSetProperties* event. This information is encoded as an OleVariant, and stored under a name you specify. Client datasets can then retrieve the information using their *GetOptionalParam* method. You can also specify that the information be included in delta packets that the client dataset sends when updating records. In this case, the client dataset may never be aware of the information, but the provider can send a round-trip message to itself.

When adding custom information in the *OnGetDataSetProperties* event, each individual attribute (sometimes called an "optional parameter") is specified using a Variant array that contains three elements: the name (a string), the value (a Variant), and a boolean flag indicating whether the information should be included in delta packets when the client applies updates. Add multiple attributes by creating a Variant array of Variant arrays. For example, the following *OnGetDataSetProperties* event handler sends two values, the time the data was provided and the total number of records in the source dataset. Only the time the data was provided is returned when client datasets apply updates:

```
procedure TMyDataModule1.Provider1GetDataSetProperties(Sender: TObject; DataSet: TDataSet;
out Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['TimeProvided', Now, True]);
  Properties[1] := VarArrayOf(['TableSize', DataSet.RecordCount, False]);
end;
```

When the client dataset applies updates, the time the original records were provided can be read in the provider's *OnUpdateData* event:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
var
  WhenProvided: TDateTime;
begin
  WhenProvided := DataSet.GetOptionalParam('TimeProvided');
  ...
end;
```

## Responding to client data requests

Usually client requests for data are handled automatically. A client dataset or XML broker requests a data packet by calling *GetRecords* (indirectly, through the *IAppServer* interface). The provider responds automatically by fetching data from the associated dataset or XML document, creating a data packet, and sending the packet to the client.

The provider has the option of editing data after it has been assembled into a data packet but before the packet is sent to the client. For example, you might want to remove records from the packet based on some criterion (such as the user's level of access), or, in a multi-tiered application, you might want to encrypt sensitive data before it is sent on to the client.

To edit the data packet before sending it on to the client, write an *OnGetData* event handler. *OnGetData* event handlers provide the data packet as a parameter in the form of a client dataset. Using the methods of this client dataset, you can edit data before it is sent to the client.

As with all method calls made through the *IAppServer* interface, the provider can communicate persistent state information with a client dataset before and after the call to *GetRecords*. This communication takes place using the *BeforeGetRecords* and *AfterGetRecords* event handlers. For a discussion of persistent state information in application servers, see "Supporting state information in remote data modules" on page 31-19.

## Responding to client update requests

A provider applies updates to database records based on a *Delta* data packet received from a client dataset or XML broker. The client requests updates by calling the *ApplyUpdates* method (indirectly, through the *IAppServer* interface).

As with all method calls made through the *IAppServer* interface, the provider can communicate persistent state information with a client dataset before and after the call to *ApplyUpdates*. This communication takes place using the *BeforeApplyUpdates* and *AfterApplyUpdates* event handlers. For a discussion of persistent state information in application servers, see "Supporting state information in remote data modules" on page 31-19.

If you are using a dataset provider, a number of additional events allow you more control:

When a dataset provider receives an update request, it generates an *OnUpdateData* event, where you can edit the Delta packet before it is written to the dataset or influence how updates are applied. After the *OnUpdateData* event, the provider writes the changes to the database or source dataset.

The provider performs the update on a record-by-record basis. Before the dataset provider applies each record, it generates a *BeforeUpdateRecord* event, which you can use to screen updates before they are applied. If an error occurs when updating a record, the provider receives an *OnUpdateError* event where it can resolve the error. Usually errors occur because the change violates a server constraint or a database record was changed by a different application subsequent to its retrieval by the provider, but prior to the client dataset's request to apply updates.

Update errors can be processed by either the dataset provider or the client dataset. When the provider is part of a multi-tiered application, it should handle all update errors that do not require user interaction to resolve. When the provider can't resolve an error condition, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered to the client dataset, and copies the unresolved records into a results data packet that it returns to the client dataset for further reconciliation.

The event handlers for all provider events are passed the set of updates as a client dataset. If your event handler is only dealing with certain types of updates, you can filter the dataset based on the update status of records. By filtering the records, your event handler does not need to sort through records it won't be using. To filter the client dataset on the update status of its records, set its *StatusFilter* property.

**Note** Applications must supply extra support when the updates are directed at a dataset that does not represent a single table. For details on how to do this, see "Applying updates to datasets that do not represent a single table" on page 30-12.

## Editing delta packets before updating the database

Before a dataset provider applies updates to the database, it generates an *OnUpdateData* event. The *OnUpdateData* event handler receives a copy of the *Delta* packet as a parameter. This is a client dataset.

In the *OnUpdateData* event handler, you can use any of the properties and methods of the client dataset to edit the *Delta* packet before it is written to the dataset. One particularly useful property is the *UpdateStatus* property. *UpdateStatus* indicates what type of modification the current record in the delta packet represents. It can have any of the values in Table 30.3.

**Table 30.3** UpdateStatus values

| Value | Description |
|---|---|
| usUnmodified | Record contents have not been changed |
| usModified | Record contents have been changed |
| usInserted | Record has been inserted |
| usDeleted | Record has been deleted |

For example, the following *OnUpdateData* event handler inserts the current date into every new record that is inserted into the database:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
  begin
    First;
    while not Eof do
    begin
      if UpdateStatus = usInserted then
      begin
        Edit;
        FieldByName('DateCreated').AsDateTime := Date;
        Post;
      end;
      Next;
    end;
end;
```

## Influencing how updates are applied

The *OnUpdateData* event also gives your dataset provider a chance to indicate how records in the delta packet are applied to the database.

By default, changes in the delta packet are written to the database using automatically generated SQL UPDATE, INSERT, or DELETE statements such as

```
UPDATE EMPLOYEES
  set EMPNO = 748, NAME = 'Smith', TITLE = 'Programmer 1', DEPT = 52
WHERE
  EMPNO = 748 and NAME = 'Smith' and TITLE = 'Programmer 1' and DEPT = 47
```

Unless you specify otherwise, all fields in the delta packet records are included in the UPDATE clause and in the WHERE clause. However, you may want to exclude some of these fields. One way to do this is to set the *UpdateMode* property of the provider. *UpdateMode* can be assigned any of the following values:

**Table 30.4** UpdateMode values

| Value | Meaning |
|---|---|
| upWhereAll | All fields are used to locate fields (the WHERE clause). |
| upWhereChanged | Only key fields and fields that are changed are used to locate records. |
| upWhereKeyOnly | Only key fields are used to locate records. |

You might, however, want even more control. For example, with the previous statement, you might want to prevent the EMPNO field from being modified by leaving it out of the UPDATE clause and leave the TITLE and DEPT fields out of the WHERE clause to avoid update conflicts when other applications have modified the data. To specify the clauses where a specific field appears, use the *ProviderFlags* property. *ProviderFlags* is a set that can include any of the values in Table 30.5

**Table 30.5** ProviderFlags values

| Value | Description |
|---|---|
| pfInWhere | The field appears in the WHERE clause of generated INSERT, DELETE, and UPDATE statements when *UpdateMode* is *upWhereAll* or *upWhereChanged*. |
| pfInUpdate | The field appears in the UPDATE clause of generated UPDATE statements. |
| pfInKey | The field is used in the WHERE clause of generated statements when *UpdateMode* is *upWhereKeyOnly*. |
| pfHidden | The field is included in records to ensure uniqueness, but can't be seen or used on the client side. |

Thus, the following *OnUpdateData* event handler allows the TITLE field to be updated and uses the EMPNO and DEPT fields to locate the desired record. If an error occurs, and a second attempt is made to locate the record based only on the key, the generated SQL looks for the EMPNO field only:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
```

```
    begin
      FieldByName('TITLE').ProviderFlags := [pfInUpdate];
      FieldByName('EMPNO').ProviderFlags := [pfInWhere, pfInKey];
      FieldByName('DEPT').ProviderFlags := [pfInWhere];
    end;
  end;
```

**Note** You can use the *UpdateFlags* property to influence how updates are applied even if you are updating to a dataset and not using dynamically generated SQL. These flags still determine which fields are used to locate records and which fields get updated.

## Screening individual updates

Immediately before each update is applied, a dataset provider receives a *BeforeUpdateRecord* event. You can use this event to edit records before they are applied, similar to the way you can use the *OnUpdateData* event to edit entire delta packets. For example, the provider does not compare BLOB fields (such as memos) when checking for update conflicts. If you want to check for update errors involving BLOB fields, you can use the *BeforeUpdateRecord* event.

In addition, you can use this event to apply updates yourself or to screen and reject updates. The *BeforeUpdateRecord* event handler lets you signal that an update has been handled already and should not be applied. The provider then skips that record, but does not count it as an update error. For example, this event provides a mechanism for applying updates to a stored procedure (which can’t be updated automatically), allowing the provider to skip any automatic processing once the record is updated from within the event handler.

## Resolving update errors on the provider

When an error condition arises as the dataset provider tries to post a record in the delta packet, an *OnUpdateError* event occurs. If the provider can’t resolve an update error, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered, and copies the unresolved records into a results data packet that it passes back to the client for further reconciliation.

In multi-tiered applications, this mechanism lets you handle any update errors you can resolve mechanically on the application server, while still allowing user interaction on the client application to correct error conditions.

The *OnUpdateError* handler gets a copy of the record that could not be changed, an error code from the database, and an indication of whether the resolver was trying to insert, delete, or update the record. The problem record is passed back in a client dataset. You should never use the data navigation methods on this dataset. However, for each field in the dataset, you can use the *NewValue*, *OldValue*, and *CurValue* properties to determine the cause of the problem and make any modifications to resolve the update error. If the *OnUpdateError* event handler can correct the problem, it sets the *Response* parameter so that the corrected record is applied.

### Applying updates to datasets that do not represent a single table

When a dataset provider generates SQL statements that apply updates directly to a database server, it needs the name of the database table that contains the records. This can be handled automatically for many datasets such as table type datasets or "live" *TQuery* components. Automatic updates are a problem however, if the provider must apply updates to the data underlying a stored procedure with a result set or a multi-table query. There is no easy way to obtain the name of the table to which updates should be applied.

If the query or stored procedure is a BDE-enabled dataset (*TQuery* or *TStoredProc*) and it has an associated update object, the provider uses the update object. However, if there is no update object, you can supply the table name programmatically in an *OnGetTableName* event handler. Once an event handler supplies the table name, the provider can generate appropriate SQL statements to apply updates.

Supplying a table name only works if the target of the updates is a single database table (that is, only the records in one table need to be updated). If the update requires making changes to multiple underlying database tables, you must explicitly apply the updates in code using the *BeforeUpdateRecord* event of the provider. Once this event handler has applied an update, you can set the event handler's *Applied* parameter to *True* so that the provider does not generate an error.

**Note** If the provider is associated with a BDE-enabled dataset, you can use an update object in the *BeforeUpdateRecord* event handler to apply updates using customized SQL statements. See "Using update objects to update a dataset" on page 26-40 for details.

## Responding to client-generated events

Provider components implement a general-purpose event that lets you create your own calls from client datasets directly to the provider. This is the *OnDataRequest* event.

*OnDataRequest* is not part of the normal functioning of the provider. It is simply a hook to allow your client datasets to communicate directly with providers. The event handler takes an OleVariant as an input parameter and returns an OleVariant. By using OleVariants, the interface is sufficiently general to accommodate almost any information you want to pass to or from the provider.

To generate an *OnDataRequest* event, the client application calls the *DataRequest* method of the client dataset.

## Handling server constraints

Most relational database management systems implement constraints on their tables to enforce data integrity. A constraint is a rule that governs data values in tables and columns, or that governs data relationships across columns in different tables. For example, most SQL-92 compliant relational databases support the following constraints:

- NOT NULL, to guarantee that a value supplied to a column has a value.
- NOT NULL UNIQUE, to guarantee that column value has a value and does not duplicate any other value already in that column for another record.
- CHECK, to guarantee that a value supplied to a column falls within a certain range, or is one of a limited number of possible values.
- CONSTRAINT, a table-wide check constraint that applies to multiple columns.
- PRIMARY KEY, to designate one or more columns as the table's primary key for indexing purposes.
- FOREIGN KEY, to designate one or more columns in a table that reference another table.

**Note** This list is not exclusive. Your database server may support some or all of these constraints in part or in whole, and may support additional constraints. For more information about supported constraints, see your server documentation.

Database server constraints obviously duplicate many kinds of data checks that traditional desktop database applications manage. You can take advantage of server constraints in multi-tiered database applications without having to duplicate the constraints in application server or client application code.

If the provider is working with a BDE-enabled dataset, the *Constraints* property lets you replicate and apply server constraints to data passed to and received from client datasets. When *Constraints* is *True* (the default), server constraints stored in the source dataset are included in data packets and affect client attempts to update data.

**Important** Before the provider can pass constraint information on to client datasets, it must retrieve the constraints from the database server. To import database constraints from the server, use SQL Explorer to import the database server's constraints and default expressions into the Data Dictionary. Constraints and default expressions in the Data Dictionary are automatically made available to BDE-enabled datasets.

There may be times when you do not want to apply server constraints to data sent to a client dataset. For example, a client dataset that receives data in packets and permits local updating of records prior to fetching more records may need to disable some server constraints that might be triggered because of the temporarily incomplete set of data. To prevent constraint replication from the provider to a client dataset, set *Constraints* to *False*. Note that client datasets can disable and enable constraints using the *DisableConstraints* and *EnableConstraints* methods. For more information about enabling and disabling constraints from the client dataset, see "Handling constraints from the server" on page 29-30.

Chapter

31

# Creating multi-tiered applications

This chapter describes how to create a multi-tiered, client/server database application. A multi-tiered client/server application is partitioned into logical units, called tiers, which run in conjunction on separate machines. Multi-tiered applications share data and communicate with one another over a local-area network or even over the Internet. They provide many benefits, such as centralized business logic and thin client applications.

In its simplest form, sometimes called the "three-tiered model," a multi-tiered application is partitioned into thirds:

- **Client application**: provides a user interface on the user's machine.
- **Application server**: resides in a central networking location accessible to all clients and provides common data services.
- **Remote database server**: provides the relational database management system (RDBMS).

In this three-tiered model, the application server manages the flow of data between clients and the remote database server, so it is sometimes called a "data broker." You usually only create the application server and its clients, although, if you are really ambitious, you could create your own database back end as well.

In more complex multi-tiered applications, additional services reside between a client and a remote database server. For example, there might be a security services broker to handle secure Internet transactions, or bridge services to handle sharing of data with databases on other platforms.

Support for developing multi-tiered applications is an extension of the way client datasets communicate with a provider component using transportable data packets. This chapter focuses on creating a three-tiered database application. Once you understand how to create and manage a three-tiered application, you can create and add additional service layers based on your needs.

## Advantages of the multi-tiered database model

The multi-tiered database model breaks a database application into logical pieces. The client application can focus on data display and user interactions. Ideally, it knows nothing about how the data is stored or maintained. The application server (middle tier) coordinates and processes requests and updates from multiple clients. It handles all the details of defining datasets and interacting with the database server.

The advantages of this multi-tiered model include the following:

- **Encapsulation of business logic in a shared middle tier**. Different client applications all access the same middle tier. This allows you to avoid the redundancy (and maintenance cost) of duplicating your business rules for each separate client application.
- **Thin client applications**. Your client applications can be written to make a small footprint by delegating more of the processing to middle tiers. Not only are client applications smaller, but they are easier to deploy because they don't need to worry about installing, configuring, and maintaining the database connectivity software (such as the database server's client-side software). Thin client applications can be distributed over the Internet for additional flexibility.
- **Distributed data processing**. Distributing the work of an application over several machines can improve performance because of load balancing, and allow redundant systems to take over when a server goes down.
- **Increased opportunity for security**. You can isolate sensitive functionality into tiers that have different access restrictions. This provides flexible and configurable levels of security. Middle tiers can limit the entry points to sensitive material, allowing you to control access more easily. If you are using HTTP or COM+, you can take advantage of the security models they support.

## Understanding multi-tiered database applications

Multi-tiered applications use the components on the DataSnap page, the Data Access page, and possibly the WebServices page of the Component palette, plus a remote data module that is created by a wizard on the Multitier or WebServices page of the New Items dialog. They are based on the ability of provider components to package data into transportable data packets and handle updates received as transportable delta packets.

The components needed for a multi-tiered application are described in Table 31.1:

**Table 31.1** Components used in multi-tiered applications

| Component | Description |
|---|---|
| Remote data modules | Specialized data modules that can act as a COM Automation server or implement a Web Service to give client applications access to any providers they contain. Used on the application server. |
| Provider component | A data broker that provides data by creating data packets and resolves client updates. Used on the application server. |
| Client dataset component | A specialized dataset that uses midas.dll or midaslib.dcu to manage data stored in data packets. The client dataset is used in the client application. It caches updates locally, and applies them in delta packets to the application server. |
| Connection components | A family of components that locate the server, form connections, and make the *IAppServer* interface available to client datasets. Each connection component is specialized to use a particular communications protocol. |

The provider and client dataset components require midas.dll or midaslib.dcu, which manages datasets stored as data packets. (Note that, because the provider is used on the application server and the client dataset is used on the client application, if you are using midas.dll, you must deploy it on both application server and client application.)

If you are using BDE-enabled datasets, the application server may also require SQL Explorer to help in database administration and to import server constraints into the Data Dictionary so that they can be checked at any level of the multi-tiered application.

**Note** You must purchase server licenses for deploying your application server.

An overview of the architecture into which these components fit is described in "Using a multi-tiered architecture" on page 19-13.

## Overview of a three-tiered application

The following numbered steps illustrate a normal sequence of events for a provider-based three-tiered application:

**1** A user starts the client application. The client connects to the application server (which can be specified at design time or runtime). If the application server is not already running, it starts. The client receives an *IAppServer* interface for communicating with the application server.

**2** The client requests data from the application server. A client may request all data at once, or may request chunks of data throughout the session (fetch on demand).

3. The application server retrieves the data (first establishing a database connection, if necessary), packages it for the client, and returns a data packet to the client. Additional information, (for example, field display characteristics) can be included in the metadata of the data packet. This process of packaging data into data packets is called “providing.”
4. The client decodes the data packet and displays the data to the user.
5. As the user interacts with the client application, the data is updated (records are added, deleted, or modified). These modifications are stored in a change log by the client.
6. Eventually the client applies its updates to the application server, usually in response to a user action. To apply updates, the client packages its change log and sends it as a data packet to the server.
7. The application server decodes the package and posts updates (in the context of a transaction if appropriate). If a record can’t be posted (for example, because another application changed the record after the client requested it and before the client applied its updates), the application server either attempts to reconcile the client’s changes with the current data, or saves the records that could not be posted. This process of posting records and caching problem records is called “resolving.”
8. When the application server finishes the resolving process, it returns any unposted records to the client for further resolution.
9. The client reconciles unresolved records. There are many ways a client can reconcile unresolved records. Typically the client attempts to correct the situation that prevented records from being posted or discards the changes. If the error situation can be rectified, the client applies updates again.
10. The client refreshes its data from the server.

## The structure of the client application

To the end user, the client application of a multi-tiered application looks and behaves no differently than a two-tiered application that uses cached updates. User interaction takes place through standard data-aware controls that display data from a *TClientDataSet* component. For detailed information about using the properties, events, and methods of client datasets, see Chapter 29, “Using client datasets.”

*TClientDataSet* fetches data from and applies updates to a provider component, just as in two-tiered applications that use a client dataset with an external provider. For details about providers, see Chapter 30, “Using provider components.” For details about client dataset features that facilitate its communication with a provider, see “Using a client dataset with a provider” on page 29-24.

The client dataset communicates with the provider through the *IAppServer* interface. It gets this interface from a connection component. The connection component establishes the connection to the application server. Different connection components are available for using different communications protocols. These connection components are summarized in the following table:

**Table 31.2** Connection components

| Component | Protocol |
|---|---|
| TDCOMConnection | DCOM |
| TSocketConnection | Windows sockets (TCP/IP) |
| TWebConnection | HTTP |
| TSOAPConnection | SOAP (HTTP and XML) |

**Note** The DataSnap page of the Component palette also includes a connection component that does not connect to an application server at all, but instead supplies an *IAppServer* interface for client datasets to use when communicating with providers in the same application. This component, *TLocalConnection*, is not required, but makes it easier to scale up to a multi-tiered application later.

For more information about using connection components, see "Connecting to the application server" on page 31-23.

## The structure of the application server

When you set up and run an application server, it does not establish any connection with client applications. Rather, client applications initiate and maintain the connection. The client application uses a connection component to connect to the application server, and uses the interface of the application server to communicate with a selected provider. All of this happens automatically, without your having to write code to manage incoming requests or supply interfaces.

The basis of an application server is a remote data module, which is a specialized data module that supports the *IAppServer* interface (for application servers that also function as a Web Service, the remote data module supports the *IAppServerSOAP* interface as well, and uses it in preference to *IAppServer*.) Client applications use the remote data module's interface to communicate with providers on the application server. When the remote data module uses *IAppServerSOAP*, the connection component adapts this to an *IAppServer* interface that client datasets can use.

There are three types of remote data modules:

- **TRemoteDataModule**: This is a dual-interface Automation server. Use this type of remote data module if clients use DCOM, HTTP, sockets, or OLE to connect to the application server, unless you want to install the application server with COM+.
- **TMTSDataModule**: This is a dual-interface Automation server. Use this type of remote data module if you are creating the application server as an Active Library (.DLL) that is installed with COM+ (or MTS). You can use MTS remote data modules with DCOM, HTTP, sockets, or OLE.
- **TSoapDataModule**: This is a data module that implements an *IAppServerSOAP* interface in a Web Service application. Use this type of remote data module to provide data to clients that access data as a Web Service.

**Note** If the application server is to be deployed under COM+ (or MTS), the remote data module includes events for when the application server is activated or deactivated. This allows it to acquire database connections when activated and release them when deactivated.

### The contents of the remote data module

As with any data module, you can include any nonvisual component in the remote data module. There are certain components, however, that you must include:

- If the remote data module is exposing information from a database server, it must include a dataset component to represent the records from that database server. Other components, such as a database connection component of some type, may be required to allow the dataset to interact with a database server. For information about datasets, see Chapter 24, "Understanding datasets." For information about database connection components, see Chapter 23, "Connecting to databases."

  For every dataset that the remote data module exposes to clients, it must include a dataset provider. A dataset provider packages data into data packets that are sent to client datasets and applies updates received from client datasets back to a source dataset or a database server. For more information about dataset providers, see Chapter 30, "Using provider components."
- For every XML document that the remote data module exposes to clients, it must include an XML provider. An XML provider acts like a dataset provider, except that it fetches data from and applies updates to an XML document rather than a database server. For more information about XML providers, see "Using an XML document as the source for a provider" on page 32-8.

**Note** Do not confuse database connection components, which connect datasets to a database server, with the connection components used by client applications in a multi-tiered application. The connection components in multi-tiered applications can be found on the DataSnap page or WebServices page of the Component palette.

## Using transactional data modules

You can write an application server that takes advantage of special services for distributed applications that are supplied by COM+ (under Windows 2000 and later) or MTS (before Windows 2000). To do so, create a transactional data module instead of an ordinary remote data module.

When you use a transactional data module, your application can take advantage of the following special services:

- **Security.** COM+ (or MTS) provides role-based security for your application server. Clients are assigned roles, which determine how they can access the MTS data module's interface. The MTS data module implements the *IsCallerInRole* method, which you lets you check the role of the currently connected client and conditionally allow certain functions based on that role. For more information about COM+ security, see "Role-based security" on page 46-15.
- **Database handle pooling.** Transactional data modules automatically pool database connections that are made via ADO or (if you are using MTS and turn on MTS POOLING) the BDE. When one client is finished with a database connection, another client can reuse it. This cuts down on network traffic, because your middle tier does not need to log off of the remote database server and then log on again. When pooling database handles, your database connection component should set its *KeepConnection* property to *False*, so that your application maximizes the sharing of connections. For more information about pooling database handles, see "Database resource dispensers" on page 46-6.
- **Transactions**. When using a transactional data module, you can provide enhanced transaction support beyond that available with a single database connection. Transactional data modules can participate in transactions that span multiple databases, or include functions that do not involve databases at all. For more information about the transaction support provided by transactional objects such as transactional data modules, see "Managing transactions in multi-tiered applications" on page 31-17.
- **Just-in-time activation and as-soon-as-possible deactivation.** You can write your server so that remote data module instances are activated and deactivated on an as-needed basis. When using just-in-time activation and as-soon-as-possible deactivation, your remote data module is instantiated only when it is needed to handle client requests. This prevents it from tying up resources such as database handles when they are not in use.

  Using just-in-time activation and as-soon-as-possible deactivation provides a middle ground between routing all clients through a single remote data module instance, and creating a separate instance for every client connection. With a single remote data module instance, the application server must handle all database calls through a single database connection. This acts as a bottleneck, and can impact performance when there are many clients. With multiple instances of the remote data module, each instance can maintain a separate database connection, thereby avoiding the need to serialize database access. However, this monopolizes resources because other clients can't use the database connection while it is associated with another client's remote data module.

To take advantage of transactions, just-in-time activation, and as-soon-as-possible deactivation, remote data module instances must be stateless. This means you must provide additional support if your client relies on state information. For example, the client must pass information about the current record when performing incremental fetches. For more information about state information and remote data modules in multi-tiered applications, see "Supporting state information in remote data modules" on page 31-19.

By default, all automatically generated calls to a transactional data module are transactional (that is, they assume that when the call exits, the data module can be deactivated and any current transactions committed or rolled back). You can write a transactional data module that depends on persistent state information by setting the *AutoComplete* property to *False*, but it will not support transactions, just-in-time activation, or as-soon-as-possible deactivation unless you use a custom interface.

**Warning** Application servers containing transactional data modules should not open database connections until the data module is activated. While developing your application, be sure that all datasets are not active and the database is not connected before running your application. In the application itself, add code to open database connections when the data module is activated and close them when it is deactivated.

### Pooling remote data modules

Object pooling allows you to create a cache of remote data modules that are shared by their clients, thereby conserving resources. How this works depends on the type of remote data module and on the connection protocol.

If you are creating a transactional data module that will be installed to COM+, you can use the COM+ Component Manager to install the application server as a pooled object. See "Object pooling" on page 46-8 for details.

Even if you are not using a transactional data module, you can take advantage of object pooling if the connection is formed using *TWebConnection*. Under this second type of object pooling, you limit the number of instances of your remote data module that are created. This limits the number of database connections that you must hold, as well as any other resources used by the remote data module.

When the Web Server application (which passes calls to your remote data module) receives client requests, it passes them on to the first available remote data module in the pool. If there is no available remote data module, it creates a new one (up to a maximum number that you specify). This provides a middle ground between routing all clients through a single remote data module instance (which can act as a bottleneck), and creating a separate instance for every client connection (which can consume many resources).

If a remote data module instance in the pool does not receive any client requests for a while, it is automatically freed. This prevents the pool from monopolizing resources unless they are used.

To set up object pooling when using a Web connection (HTTP), your remote data module must override the *UpdateRegistry* method. In the overridden method, call *RegisterPooled* when the remote data module registers and *UnregisterPooled* when the remote data module unregisters. When using either method of object pooling, your remote data module must be stateless. This is because a single instance potentially handles requests from several clients. If it relied on persistent state information, clients could interfere with each other. See "Supporting state information in remote data modules" on page 31-19 for more information on how to ensure that your remote data module is stateless.

## Choosing a connection protocol

Each communications protocol you can use to connect your client applications to the application server provides its own unique benefits. Before choosing a protocol, consider how many clients you expect, how you are deploying your application, and future development plans.

### Using DCOM connections

DCOM provides the most direct approach to communication, requiring no additional runtime applications on the server.

DCOM provides the only approach that lets you use security services when writing a transactional data module. These security services are based on assigning roles to the callers of transactional objects. When using DCOM, DCOM identifies the caller to the system that calls your application server (COM+ or MTS). Therefore, it is possible to accurately determine the role of the caller. When using other protocols, however, there is a runtime executable, separate from the application server, that receives client calls. This runtime executable makes COM calls into the application server on behalf of the client. Because of this, it is impossible to assign roles to separate clients: The runtime executable is, effectively, the only client. For more information about security and transactional objects, see "Role-based security" on page 46-15.

### Using Socket connections

TCP/IP Sockets let you create lightweight clients. For example, if you are writing a Web-based client application, you can't be sure that client systems support DCOM. Sockets provide a lowest common denominator that you know will be available for connecting to the application server. For more information about sockets, see Chapter 39, "Working with sockets."

Instead of instantiating the remote data module directly from the client (as happens with DCOM), sockets use a separate application on the server (ScktSrvr.exe), which accepts client requests and instantiates the remote data module using COM. The connection component on the client and ScktSrvr.exe on the server are responsible for marshaling *IAppServer* calls.

**Note** ScktSrvr.exe can run as an NT service application. Register it with the Service manager by starting it using the -install command line option. You can unregister it using the -uninstall command line option.

Before you can use a socket connection, the application server must register its availability to clients using a socket connection. By default, all new remote data modules automatically register themselves by adding a call to *EnableSocketTransport* in the *UpdateRegistry* method. You can remove this call to prevent socket connections to your application server.

**Note** Because older servers did not add this registration, you can disable the check for whether an application server is registered by unchecking the Connections | Registered Objects Only menu item on ScktSrvr.exe.

When using sockets, there is no protection on the server against client systems failing before they release a reference to interfaces on the application server. While this results in less message traffic than when using DCOM (which sends periodic keep-alive messages), this can result in an application server that can't release its resources because it is unaware that the client has gone away.

### Using Web connections

HTTP lets you create clients that can communicate with an application server that is protected by a firewall. HTTP messages provide controlled access to internal applications so that you can distribute your client applications safely and widely. Like socket connections, HTTP messages provide a lowest common denominator that you know will be available for connecting to the application server. For more information about HTTP messages, see Chapter 33, "Creating Internet server applications."

Instead of instantiating the remote data module directly from the client (as happens with DCOM), HTTP-based connections use a Web server application on the server (httpsrvr.dll) that accepts client requests and instantiates the remote data module using COM. Because of this, they are also called Web connections. The connection component on the client and httpsrvr.dll on the server are responsible for marshaling *IAppServer* calls.

Web connections can take advantage of the SSL security provided by wininet.dll (a library of Internet utilities that runs on the client system). Once you have configured the Web server on the server system to require authentication, you can specify the user name and password using the properties of the Web connection component.

As an additional security measure, the application server must register its availability to clients using a Web connection. By default, all new remote data modules automatically register themselves by adding a call to *EnableWebTransport* in the *UpdateRegistry* method. You can remove this call to prevent Web connections to your application server.

Web connections can take advantage of object pooling. This allows your server to create a limited pool of remote data module instances that are available for client requests. By pooling the remote data modules, your server does not consume the resources for the data module and its database connection except when they are needed. For more information on object pooling, see "Pooling remote data modules" on page 31-8.

Unlike most other connection components, you can’t use callbacks when the connection is formed via HTTP.

### Using SOAP connections

SOAP is the protocol that underlies the built-in support for Web Service applications. SOAP marshals method calls using an XML encoding. SOAP connections use HTTP as a transport protocol.

SOAP connections have the advantage that they work in cross-platform applications because they are supported on both the Windows and Linux. Because SOAP connections use HTTP, they have the same advantages as Web connections: HTTP provides a lowest common denominator that you know is available on all clients, and clients can communicate with an application server that is protected by a “firewall.” For more information about using SOAP to distribute applications, see Chapter 38, “Using Web Services.”

As with HTTP connections, you can’t use callbacks when the connection is formed via SOAP.

## Building a multi-tiered application

The general steps for creating a multi-tiered database application are

1. Create the application server.
2. Register or install the application server.
3. Create a client application.

The order of creation is important. You should create and run the application server before you create a client. At design time, you can then connect to the application server to test your client. You can, of course, create a client without specifying the application server at design time, and only supply the server name at runtime. However, doing so prevents you from seeing if your application works as expected when you code at design time, and you will not be able to choose servers and providers using the Object Inspector.

**Note** If you are not creating the client application on the same system as the server, and you are using a DCOM connection, you may want to register the application server on the client system. This makes the connection component aware of the application server at design time so that you can choose server names and provider names from a drop-down list in the Object Inspector. (If you are using a Web connection, SOAP connection, or socket connection, the connection component fetches the names of registered providers from the server machine.)

## Creating the application server

You create an application server very much as you create most database applications. The major difference is that the application server uses a remote data module.

To create an application server, follow these steps:

1. Start a new project:

   - If you are using SOAP as a transport protocol, this should be a new Web Service application. Choose File | New | Other, and on the WebServices page of the new items dialog, choose SOAP Server application. Select the type of Web Server you want to use, and when prompted whether you want to define a new interface for the SOAP module, say no.
   - For any other transport protocol, you need only choose File | New | Application.

   Save the new project.

2. Add a new remote data module to the project. From the main menu, choose File | New | Other, and on the MultiTier or WebServices page of the new items dialog, select

   - **Remote Data Module** if you are creating a COM Automation server that clients access using DCOM, HTTP, or sockets.
   - **Transactional Data Module** if you are creating a remote data module that runs under COM+ (or MTS). Connections can be formed using DCOM, HTTP, or sockets. However, only DCOM supports the security services.
   - **SOAP Server Data Module** if you are creating a SOAP server in a Web Service application.

   For more detailed information about setting up a remote data module, see "Setting up the remote data module" on page 31-13.

   **Note** Remote data modules are more than simple data modules. The SOAP data module implements an invokable interface in a Web Service application. Other data modules are COM Automation objects.

3. Place the appropriate dataset components on the data module and set them up to access the database server.

4. Place a *TDataSetProvider* component on the data module for each dataset you want to expose to clients. This provider is required for brokering client requests and packaging data. Set the *DataSet* property for each provider to the name of the dataset to access. You can set additional properties for the provider. See Chapter 30, "Using provider components" for more detailed information about setting up a provider.

   If you are working with data from XML documents, you can use a *TXMLTransformProvider* component instead of a dataset and *TDataSetProvider* component. When using *TXMLTransformProvider*, set the *XMLDataFile* property to specify the XML document from which data is provided and to which updates are applied.

5. Write application server code to implement events, shared business rules, shared data validation, and shared security. When writing this code, you may want to
   - Extend the application server's interface to provide additional ways for the client application to call the server. Extending the application server's interface is described in "Extending the application server's interface" on page 31-16.
   - Provide transaction support beyond the transactions automatically created when applying updates. Transaction support in multi-tiered database applications is described in "Managing transactions in multi-tiered applications" on page 31-17.
   - Create master/detail relationships between the datasets in your application server. Master/detail relationships are described in "Supporting master/detail relationships" on page 31-18.
   - Ensure your application server is stateless. Handling state information is described in "Supporting state information in remote data modules" on page 31-19.
   - Divide your application server into multiple remote data modules. Using multiple remote data modules is described in "Using multiple remote data modules" on page 31-21.
6. Save, compile, and register or install the application server. Registering an application server is described in "Registering the application server" on page 31-22.
7. If your server application does not use DCOM or SOAP, you must install the runtime software that receives client messages, instantiates the remote data module, and marshals interface calls.
   - For TCP/IP sockets this is a socket dispatcher application, Scktsrvr.exe.
   - For HTTP connections this is httpsrvr.dll, an ISAPI/NSAPI DLL that must be installed with your Web server.

## Setting up the remote data module

When you create the remote data module, you must provide certain information that indicates how it responds to client requests. This information varies, depending on the type of remote data module. See "The structure of the application server" on page 31-5 for information on what type of remote data module you need.

### Configuring TRemoteDataModule

To add a *TRemoteDataModule* component to your application, choose File|New|Other and select Remote Data Module from the Multitier page of the new items dialog. You will see the Remote Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TRemoteDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TRemoteDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

**Note** You can add your own properties and methods to the new interface. For more information, see "Extending the application server's interface" on page 31-16.

You must specify the threading model in the Remote Data Module wizard. You can choose Single-threaded, Apartment-threaded, Free-threaded, or Both.

- If you choose Single-threaded, COM ensures that only one client request is serviced at a time. You do not need to worry about client requests interfering with each other.
- If you choose Apartment-threaded, COM ensures that any instance of your remote data module services one request at a time. When writing code in an Apartment-threaded library, you must guard against thread conflicts if you use global variables or objects not contained in the remote data module. This is the recommended model if you are using BDE-enabled datasets. (Note that you will need a session component with its *AutoSessionName* property set to *True* to handle threading issues on BDE-enabled datasets).
- If you choose Free-threaded, your application can receive simultaneous client requests on several threads. You are responsible for ensuring your application is thread-safe. Because multiple clients can access your remote data module simultaneously, you must guard your instance data (properties, contained objects, and so on) as well as global variables. This is the recommended model if you are using ADO datasets.
- If you choose Both, your library works the same as when you choose Free-threaded, with one exception: all callbacks (calls to client interfaces) are serialized for you.
- If you choose Neutral, the remote data module can receive simultaneous calls on separate threads, as in the Free-threaded model, but COM guarantees that no two threads access the same method at the same time.

If you are creating an EXE, you must also specify what type of instancing to use. You can choose Single instance or Multiple instance (Internal instancing applies only if the client code is part of the same process space.)

- If you choose Single instance, each client connection launches its own instance of the executable. That process instantiates a single instance of the remote data module, which is dedicated to the client connection.
- If you choose Multiple instance, a single instance of the application (process) instantiates all remote data modules created for clients. Each remote data module is dedicated to a single client connection, but they all share the same process space.

### Configuring TMTSDataModule

To add a *TMTSDataModule* component to your application, choose File | New | Other and select Transactional Data Module from the Multitier page of the new items dialog. You will see the Transactional Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TMTSDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TMTSDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

**Note** You can add your own properties and methods to your new interface. For more information, see "Extending the application server's interface" on page 31-16.

You must specify the threading model in the Transactional Data Module wizard. Choose Single, Apartment, or Both.

- If you choose Single, client requests are serialized so that your application services only one at a time. You do not need to worry about client requests interfering with each other.
- If you choose Apartment, the system ensures that any instance of your remote data module services one request at a time, and calls always use the same thread. You must guard against thread conflicts if you use global variables or objects not contained in the remote data module. Instead of using global variables, you can use the shared property manager. For more information on the shared property manager, see "Shared property manager" on page 46-6.
- If you choose Both, MTS calls into the remote data module's interface in the same way as when you choose Apartment. However, any callbacks you make to client applications are serialized, so that you don't need to worry about them interfering with each other.

**Note** The Apartment model under MTS or COM+ is different from the corresponding model under DCOM.

You must also specify the transaction attributes of your remote data module. You can choose from the following options:

- Requires a transaction. When you select this option, every time a client uses your remote data module's interface, that call is executed in the context of a transaction. If the caller supplies a transaction, a new transaction need not be created.
- Requires a new transaction. When you select this option, every time a client uses your remote data module's interface, a new transaction is automatically created for that call.
- Supports transactions. When you select this option, your remote data module can be used in the context of a transaction, but the caller must supply the transaction when it invokes the interface.
- Does not support transactions. When you select this option, your remote data module can't be used in the context of transactions.

### Configuring TSoapDataModule

To add a *TSoapDataModule* component to your application, choose File|New|Other and select SOAP Server Data Module from the WebServices page of the new items dialog. The SOAP data module wizard appears.

You must supply a class name for your SOAP data module. This is the base name of a *TSoapDataModule* descendant that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TSoapDataModule*, which implements *IMyDataServer*, a descendant of *IAppServerSOAP*.

**Note** To use *TSoapDataModule*, the new data module should be added to a Web Service application. The *IAppServerSOAP* interface is an invokable interface, which is registered in the initialization section of the new unit. This allows the invoker component in the main Web module to forward all incoming calls to your data module.

You may want to edit the definitions of the generated interface and *TSoapDataModule* descendant, adding your own properties and methods. These properties and methods are not called automatically, but client applications that request your new interface by name or GUID can use any of the properties and methods that you add.

## Extending the application server's interface

Client applications interact with the application server by creating or connecting to an instance of the remote data module. They use its interface as the basis of all communication with the application server.

You can add to your remote data module's interface to provide additional support for your client applications. This interface is a descendant of *IAppServer* and is created for you automatically by the wizard when you create the remote data module.

To add to the remote data module's interface, you can

- Choose the Add to Interface command from the Edit menu in the IDE. Indicate whether you are adding a procedure, function, or property, and enter its syntax. When you click OK, you will be positioned in the code editor on the implementation of your new interface member.
- Use the type library editor. Select the interface for your application server in the type library editor, and click the tool button for the type of interface member (method or property) that you are adding. Give your interface member a name in the Attributes page, specify parameters and type in the Parameters page, and then refresh the type library. See Chapter 41, "Working with type libraries" for more information about using the type library editor.

**Note** Neither of these approaches works if you are implementing *TSoapDataModule*. For *TSoapDataModule* descendants, you must edit the server interface directly.

When you add to a COM interface, your changes are added to your unit source code and the type library file (.TLB).

**Note** You must explicitly save the TLB file by choosing Refresh in the type library editor and then saving the changes from the IDE.

Once you have added to your remote data module's interface, locate the properties and methods that were added to your remote data module's implementation. Add code to finish this implementation by filling in the bodies of the new methods.

If you are not writing a SOAP data module, client applications call your interface extensions using the *AppServer* property of their connection component. With SOAP data modules, they call the connection component's *GetSOAPServer* method. For more information on how to call your interface extensions, see "Calling server interfaces" on page 31-28.

### Adding callbacks to the application server's interface

You can allow the application server to call your client application by introducing a callback. To do this, the client application passes an interface to one of the application server's methods, and the application server later calls this method as needed. However, if your extensions to the remote data module's interface include callbacks, you can't use an HTTP or SOAP-based connection. *TWebConnection* and *TSoapConnection* do not support callbacks. If you are using a socket-based connection, client applications must indicate whether they are using callbacks by setting the *SupportCallbacks* property. All other types of connection automatically support callbacks.

### Extending a transactional application server's interface

When using transactions or just-in-time activation, you must be sure all new methods call *SetComplete* to indicate when they are finished. This allows transactions to complete and permits the remote data module to be deactivated.

Furthermore, you can't return any values from your new methods that allow the client to communicate directly with objects or interfaces on the application server unless they provide a safe reference. If you are using a stateless MTS data module, neglecting to use a safe reference can lead to crashes because you can't guarantee that the remote data module is active. For more information on safe references, see "Passing object references" on page 46-23.

## Managing transactions in multi-tiered applications

When client applications apply updates to the application server, the provider component automatically wraps the process of applying updates and resolving errors in a transaction. This transaction is committed if the number of problem records does not exceed the *MaxErrors* value specified as an argument to the *ApplyUpdates* method. Otherwise, it is rolled back.

In addition, you can add transaction support to your server application by adding a database connection component or managing the transaction directly by sending SQL to the database server. This works the same way that you would manage transactions in a two-tiered application. For more information about this sort of transaction control, see "Managing transactions" on page 23-6.

If you have a transactional data module, you can broaden your transaction support by using COM+ (or MTS) transactions. These transactions can include any of the business logic on your application server, not just the database access. In addition, because they support two-phase commits, they can span multiple databases.

Only the BDE- and ADO-based data access components support two-phase commit. Do not use InterbaseExpress or dbExpress components if you want to have transactions that span multiple databases.

**Warning** When using the BDE, two-phase commit is fully implemented only on Oracle7 and MS-SQL databases. If your transaction involves multiple databases, and some of them are remote servers other than Oracle7 or MS-SQL, your transaction runs a small risk of only partially succeeding. Within any one database, however, you will always have transaction support.

By default, all *IAppServer* calls on a transactional data module are transactional. You need only set the transaction attribute of your data module to indicate that it must participate in transactions. In addition, you can extend the application server's interface to include method calls that encapsulate transactions that you define.

If your transaction attribute indicates that the remote data module requires a transaction, then every time a client calls a method on its interface, it is automatically wrapped in a transaction. All client calls to your application server are then enlisted in that transaction until you indicate that the transaction is complete. These calls either succeed as a whole or are rolled back.

**Note** Do not combine COM+ or MTS transactions with explicit transactions created by a database connection component or using explicit SQL commands. When your transactional data module is enlisted in a transaction, it automatically enlists all of your database calls in the transaction as well.

For more information about using COM+ (or MTS) transactions, see "MTS and COM+ transaction support" on page 46-9.

## Supporting master/detail relationships

You can create master/detail relationships between client datasets in your client application in the same way you set them up using any table-type dataset. For more information about setting up master/detail relationships in this way, see "Creating master/detail relationships" on page 24-35.

However, this approach has two major drawbacks:

- The detail table must fetch and store all of its records from the application server even though it only uses one detail set at a time. (This problem can be mitigated by using parameters. For more information, see "Limiting records with parameters" on page 29-29.)
- It is very difficult to apply updates, because client datasets apply updates at the dataset level and master/detail updates span multiple datasets. Even in a two-tiered environment, where you can use the database connection component to apply updates for multiple tables in a single transaction, applying updates in master/detail forms is tricky.

In multi-tiered applications, you can avoid these problems by using nested tables to represent the master/detail relationship. To do this when providing from datasets, set up a master/detail relationship between the datasets on the application server. Then set the *DataSet* property of your provider component to the master table. To use nested tables to represent master/detail relationships when providing from XML documents, use a transformation file that defines the nested detail sets.

When clients call the *GetRecords* method of the provider, it automatically includes the detail dataset as a DataSet field in the records of the data packet. When clients call the *ApplyUpdates* method of the provider, it automatically handles applying updates in the proper order.

## Supporting state information in remote data modules

The *IAppServer* interface, which client datasets use to communicate with providers on the application server, is mostly stateless. When an application is stateless, it does not "remember" anything that happened in previous calls by the client. This stateless quality is useful if you are pooling database connections in a transactional data module, because your application server does not need to distinguish between database connections for persistent information such as record currency. Similarly, this stateless quality is important when you are sharing remote data module instances between many clients, as occurs with just-in-time activation or object pooling. SOAP data modules must be stateless.

However, there are times when you want to maintain state information between calls to the application server. For example, when requesting data using incremental fetching, the provider on the application server must "remember" information from previous calls (the current record).

Before and after any calls to the *IAppServer* interface that the client dataset makes (AS_*ApplyUpdates*, AS_*Execute*, AS_*GetParams*, AS_*GetRecords*, or AS_*RowRequest*), it receives an event where it can send or retrieve custom state information. Similarly, before and after providers respond to these client-generated calls, they receive events where they can retrieve or send custom state information. Using this mechanism, you can communicate persistent state information between client applications and the application server, even if the application server is stateless.

For example, consider a dataset that represents the following parameterized query:

```
SELECT * from CUSTOMER WHERE CUST_NO > :MinVal ORDER BY CUST_NO
```

To enable incremental fetching in a stateless application server, you can do the following:

- When the provider packages a set of records in a data packet, it notes the value of CUST_NO on the last record in the packet:

```
TRemoteDataModule1.DataSetProvider1GetData(Sender: TObject; DataSet:
  TCustomClientDataSet);
begin
  DataSet.Last; { move to the last record }
  with Sender as TDataSetProvider do
    Tag := DataSet.FieldValues['CUST_NO']; {save the value of CUST_NO }
end;
```

- The provider sends this last CUST_NO value to the client after sending the data packet:

```
TRemoteDataModule1.DataSetProvider1AfterGetRecords(Sender: TObject;
                  var OwnerData: OleVariant);
begin
  with Sender as TDataSetProvider do
    OwnerData := Tag; {send the last value of CUST_NO }
end;
```

- On the client, the client dataset saves this last value of CUST_NO:

```
TDataModule1.ClientDataSet1AfterGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TClientDataSet do
    Tag := OwnerData; {save the last value of CUST_NO }
end;
```

- Before fetching a data packet, the client sends the last value of CUST_NO it received:

```
TDataModule1.ClientDataSet1BeforeGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TClientDataSet do
  begin
    if not Active then Exit;
    OwnerData := Tag; { Send last value of CUST_NO to application server }
  end;
end;
```

- Finally, on the server, the provider uses the last CUST_NO sent as a minimum value in the query:

```
TRemoteDataModule1.DataSetProvider1BeforeGetRecords(Sender: TObject;
                   var OwnerData: OleVariant);
begin
  if not VarIsEmpty(OwnerData) then
    with Sender as TDataSetProvider do
      with DataSet as TSQLDataSet do
      begin
        Params.ParamValues['MinVal'] := OwnerData;
        Refresh; { force the query to reexecute }
    end;
end;
```

## Using multiple remote data modules

You may want to structure your application server so that it uses multiple remote data modules. Using multiple remote data modules lets you partition your code, organizing a large application server into multiple units, where each unit is relatively self-contained.

Although you can always create multiple remote data modules on the application server that function independently, a special connection component on the DataSnap page of the Component palette provides support for a model where you have one main "parent" remote data module that dispatches connections from clients to other "child" remote data modules. This model requires that you use a COM-based application server (that is, not *TSoapDataModule*).

To create the parent remote data module, you must extend its *IAppServer* interface, adding properties that expose the interfaces of the child remote data modules. That is, for each child remote data module, add a property to the parent data module's interface whose value is the *IAppServer* interface for the child data module. The property getter should look something like the following:

```
function ParentRDM.Get_ChildRDM: IChildRDM;
begin
  if not Assigned(ChildRDMFactory) then
    ChildRDMFactory :=
      TComponentFactory.Create(ComServer, TChildRDM, Class_ChildRDM,
                               ciInternal, tmApartment);
  Result := ChildRDMFactory.CreateCOMObject(nil) as IChildRDM;
  Result.MainRDM := Self;
end;
```

For information about extending the parent remote data module's interface, see "Extending the application server's interface" on page 31-16.

**Tip** You may also want to extend the interface for each child data module, exposing the parent data module's interface, or the interfaces of the other child data modules. This lets the various data modules in your application server communicate more freely with each other.

Once you have added properties that represent the child remote data modules to the main remote data module, client applications do not need to form separate connections to each remote data module on the application server. Instead, they share a single connection to the parent remote data module, which then dispatches messages to the "child" data modules. Because each client application uses the same connection for every remote data module, the remote data modules can share a single database connection, conserving resources. For information on how child applications share a single connection, see "Connecting to an application server that uses multiple data modules" on page 31-30.

## Registering the application server

Before client applications can locate and use an application server, it must be registered or installed.

- If the application server uses DCOM, HTTP, or sockets as a communication protocol, it acts as an Automation server and must be registered like any other COM server. For information about registering a COM server, see "Registering a COM object" on page 43-17.
- If you are using a transactional data module, you do not register the application server. Instead, you install it with COM+ or MTS. For information about installing transactional objects, see "Installing transactional objects" on page 46-26.
- When the application server uses SOAP, the application must be a Web Service application. As such, it must be registered with your Web Server, so that it receives incoming HTTP messages. In addition, you need to publish a WSDL document that describes the invokable interfaces in your application. For information about exporting a WSDL document for a Web Service application, see "Generating WSDL documents for a Web Service application" on page 38-19.

## Creating the client application

In most regards, creating a multi-tiered client application is similar to creating a two-tiered client that uses a client dataset to cache updates. The major difference is that a multi-tiered client uses a connection component to establish a conduit to the application server.

To create a multi-tiered client application, start a new project and follow these steps:

1. Add a new data module to the project.
2. Place a connection component on the data module. The type of connection component you add depends on the communication protocol you want to use. See "The structure of the client application" on page 31-4 for details.
3. Set properties on your connection component to specify the application server with which it should establish a connection. To learn more about setting up the connection component, see "Connecting to the application server" on page 31-23.
4. Set the other connection component properties as needed for your application. For example, you might set the *ObjectBroker* property to allow the connection component to choose dynamically from several servers. For more information about using the connection components, see "Managing server connections" on page 31-27
5. Place as many *TClientDataSet* components as needed on the data module, and set the *RemoteServer* property for each component to the name of the connection component you placed in Step 2. For a full introduction to client datasets, see Chapter 29, "Using client datasets."

6 Set the *ProviderName* property for each *TClientDataSet* component. If your connection component is connected to the application server at design time, you can choose available application server providers from the *ProviderName* property's drop-down list.

7 Continue in the same way you would create any other database application. There are a few additional features available to clients of multi-tiered applications:

- Your application may mwant to make direct calls to the application server. "Calling server interfaces" on page 31-28 describes how to do this.
- You may want to use the special features of client datasets that support their interaction with the provider components. These are described in "Using a client dataset with a provider" on page 29-24.

## Connecting to the application server

To establish and maintain a connection to an application server, a client application uses one or more connection components. You can find these components on the DataSnap or WebServices page of the Component palette.

Use a connection component to

- Identify the protocol for communicating with the application server. Each type of connection component represents a different communication protocol. See "Choosing a connection protocol" on page 31-9 for details on the benefits and limitations of the available protocols.
- Indicate how to locate the server machine. The details of identifying the server machine vary depending on the protocol.
- Identify the application server on the server machine.
- If you are not using SOAP, identify the server using the *ServerName* or *ServerGUID* property. *ServerName* identifies the base name of the class you specify when creating the remote data module on the application server. See "Setting up the remote data module" on page 31-13 for details on how this value is specified on the server. If the server is registered or installed on the client machine, or if the connection component is connected to the server machine, you can set the *ServerName* property at design time by choosing from a drop-down list in the Object Inspector. *ServerGUID* specifies the GUID of the remote data module's interface. You can look up this value using the type library editor.

  If you are using SOAP, the server is identified in the URL you use to locate the server machine. Follow the steps in "Specifying a connection using SOAP" on page 31-26.
- Manage server connections. Connection components can be used to create or drop connections and to call application server interfaces.

Usually the application server is on a different machine from the client application, but even if the server resides on the same machine as the client application (for example, during the building and testing of the entire multi-tier application), you can still use the connection component to identify the application server by name, specify a server machine, and use the application server interface.

### Specifying a connection using DCOM

When using DCOM to communicate with the application server, client applications include a *TDCOMConnection* component for connecting to the application server. *TDCOMConnection* uses the *ComputerName* property to identify the machine on which the server resides.

When *ComputerName* is blank, the DCOM connection component assumes that the application server resides on the client machine or that the application server has a system registry entry. If you do not provide a system registry entry for the application server on the client when using DCOM, and the server resides on a different machine from the client, you must supply *ComputerName*.

**Note** Even when there is a system registry entry for the application server, you can specify *ComputerName* to override this entry. This can be especially useful during development, testing, and debugging.

If you have multiple servers that your client application can choose from, you can use the *ObjectBroker* property instead of specifying a value for *ComputerName*. For more information, see "Brokering connections" on page 31-27.

If you supply the name of a host computer or server that cannot be found, the DCOM connection component raises an exception when you try to open the connection.

### Specifying a connection using sockets

You can establish a connection to the application server using sockets from any machine that has a TCP/IP address. This method has the advantage of being applicable to more machines, but does not provide for using any security protocols. When using sockets, include a *TSocketConnection* component for connecting to the application server.

*TSocketConnection* identifies the server machine using the IP Address or host name of the server system, and the port number of the socket dispatcher program (Scktsrvr.exe) that is running on the server machine. For more information about IP addresses and port values, see "Describing sockets" on page 39-4.

Three properties of *TSocketConnection* specify this information:

- *Address* specifies the IP Address of the server.
- *Host* specifies the host name of the server.
- *Port* specifies the port number of the socket dispatcher program on the application server.

*Address* and *Host* are mutually exclusive. Setting one unsets the value of the other. For information on which one to use, see "Describing the host" on page 39-4.

If you have multiple servers that your client application can choose from, you can use the *ObjectBroker* property instead of specifying a value for *Address* or *Host*. For more information, see "Brokering connections" on page 31-27.

By default, the value of *Port* is 211, which is the default port number of the socket dispatcher program that forwards incoming messages to your application server. If the socket dispatcher has been configured to use a different port, set the *Port* property to match that value.

**Note** You can configure the port of the socket dispatcher while it is running by right-clicking the Borland Socket Server tray icon and choosing Properties.

Although socket connections do not provide for using security protocols, you can customize the socket connection to add your own encryption. To do this

1 Create a COM object that supports the *IDataIntercept* interface. This is an interface for encrypting and decrypting data.
2 Use *TPacketInterceptFactory* as the class factory for this object. If you are using a wizard to create the COM object in step 1, replace the line in the initialization section that says `TComponentFactory.Create(...)` with `TPacketInterceptFactory.Create(...)`.
3 Register your new COM server on the client machine.
4 Set the *InterceptName* or *InterceptGUID* property of the socket connection component to specify this COM object. If you used *TPacketInterceptFactory* in step 2, your COM server appears in the drop-down list of the Object Inspector for the *InterceptName* property.
5 Finally, right click the Borland Socket Server tray icon, choose Properties, and on the properties tab set the Intercept Name or Intercept GUID to the ProgId or GUID for the interceptor.

This mechanism can also be used for data compression and decompression.

### Specifying a connection using HTTP

You can establish a connection to the application server using HTTP from any machine that has a TCP/IP address. Unlike sockets, however, HTTP allows you to take advantage of SSL security and to communicate with a server that is protected behind a firewall. When using HTTP, include a *TWebConnection* component for connecting to the application server.

The Web connection component establishes a connection to the Web server application (httpsrvr.dll), which in turn communicates with the application server. *TWebConnection* locates httpsrvr.dll using a Uniform Resource Locator (URL). The URL specifies the protocol (http or, if you are using SSL security, https), the host name for the machine that runs the Web server and httpsrvr.dll, and the path to the Web server application (httpsrvr.dll). Specify this value using the *URL* property.

**Note** When using *TWebConnection*, wininet.dll must be installed on the client machine. If you have IE3 or higher installed, wininet.dll can be found in the Windows system directory.

If the Web server requires authentication, or if you are using a proxy server that requires authentication, you must set the values of the *UserName* and *Password* properties so that the connection component can log on.

If you have multiple servers that your client application can choose from, you can use the *ObjectBroker* property instead of specifying a value for *URL*. For more information, see “Brokering connections” on page 31-27.

### Specifying a connection using SOAP

You can establish a connection to a SOAP application server using the *TSoapConnection* component. *TSoapConnection* is very similar to *TWebConnection*, because it also uses HTTP as a transport protocol. Thus, you can use *TSoapConnection* from any machine that has a TCP/IP address, and it can take advantage of SSL security and to communicate with a server that is protected by a firewall.

The SOAP connection component establishes a connection to a Web Service provider that implements the *IAppServerSOAP* or *IAppServer* interface. (The *UseSOAPAdapter* property specifies which interface it expects the server to support.) If the server implements the *IAppServerSOAP* interface, *TSoapConnection* converts that interface to an *IAppServer* interface for client datasets. *TSoapConnection* locates the Web Server application using a Uniform Resource Locator (URL). The URL specifies the protocol (http or, if you are using SSL security, https), the host name for the machine that runs the Web server, the name of the Web Service application, and a path that matches the path name of the *THTTPSoapDispatcher* on the application server. Specify this value using the *URL* property.

By default, *TSOAPConnection* automatically looks for an *IAppServerSOAP* (or *IAppServer*) interface. If the server includes more than one remote data module, you must indicate the target data module’s interface (an *IAppServerSOAP* descendant) so that *TSOAPConnection* can identify the remote data module you want to use. There are two ways to do this:

- Set the *SOAPServerIID* property to indicate the interface of the target remote data module. This method works for any server that implements an *IAppServerSOAP* descendant. *SOAPServerIID* identifies the target interface by its GUID. At runtime, you can use the interface name, and the compiler automatically extracts the GUID. However, at design time, in the Object Inspector, you must specify the GUID string.
- If the server is written using the Delphi language, you can simply include the name of the SOAP data module’s interface following a slash at the end of the path portion of the URL. This lets you specify the interface by name rather than GUID, but is only available when both client and server are written in Delphi.

**Tip** The first approach, using the *SOAPServerIID* method, has the added advantage that it lets you call extensions to the remote data module’s interface.

If you are using a proxy server, you must indicate the name of the proxy server using the *Proxy* property. If that proxy requires authentication, you must also set the values of the *UserName* and *Password* properties so that the connection component can log on.

**Note** When using *TSoapConnection*, wininet.dll must be installed on the client machine. If you have IE3 or higher installed, wininet.dll can be found in the Windows system directory.

### Brokering connections

If you have multiple COM-based servers that your client application can choose from, you can use an Object Broker to locate an available server system. The object broker maintains a list of servers from which the connection component can choose. When the connection component needs to connect to an application server, it asks the Object Broker for a computer name (or IP address, host name, or URL). The broker supplies a name, and the connection component forms a connection. If the supplied name does not work (for example, if the server is down), the broker supplies another name, and so on, until a connection is formed.

Once the connection component has formed a connection with a name supplied by the broker, it saves that name as the value of the appropriate property (*ComputerName*, *Address*, *Host*, *RemoteHost*, or *URL*). If the connection component closes the connection later, and then needs to reopen the connection, it tries using this property value, and only requests a new name from the broker if the connection fails.

Use an Object Broker by specifying the *ObjectBroker* property of your connection component. When the *ObjectBroker* property is set, the connection component does not save the value of *ComputerName*, *Address*, *Host*, *RemoteHost*, or *URL* to disk.

**Note** You can not use the *ObjectBroker* property with SOAP connections.

## Managing server connections

The main purpose of connection components is to locate and connect to the application server. Because they manage server connections, you can also use connection components to call the methods of the application server's interface.

### Connecting to the server

To locate and connect to the application server, you must first set the properties of the connection component to identify the application server. This process is described in "Connecting to the application server" on page 31-23. Before opening the connection, any client datasets that use the connection component to communicate with the application server should indicate this by setting their *RemoteServer* property to specify the connection component.

The connection is opened automatically when client datasets try to access the application server. For example, setting the *Active* property of the client dataset to *True* opens the connection, as long as the *RemoteServer* property has been set.

If you do not link any client datasets to the connection component, you can open the connection by setting the *Connected* property of the connection component to *True*.

Before a connection component establishes a connection to an application server, it generates a *BeforeConnect* event. You can perform any special actions prior to connecting in a *BeforeConnect* handler that you code. After establishing a connection, the connection component generates an *AfterConnect* event for any special actions.

### Dropping or changing a server connection

A connection component drops a connection to the application server when you

- set the *Connected* property to *False*.
- free the connection component. A connection object is automatically freed when a user closes the client application.
- change any of the properties that identify the application server (*ServerName*, *ServerGUID*, *ComputerName*, and so on). Changing these properties allows you to switch among available application servers at runtime. The connection component drops the current connection and establishes a new one.

**Note** Instead of using a single connection component to switch among available application servers, a client application can instead have more than one connection component, each of which is connected to a different application server.

Before a connection component drops a connection, it automatically calls its *BeforeDisconnect* event handler, if one is provided. To perform any special actions prior to disconnecting, write a *BeforeDisconnect* handler. Similarly, after dropping the connection, the *AfterDisconnect* event handler is called. If you want to perform any special actions after disconnecting, write an *AfterDisconnect* handler.

## Calling server interfaces

Applications do not need to call the *IAppServer* or *IAppServerSOAP* interface directly because the appropriate calls are made automatically when you use the properties and methods of the client dataset. However, while it is not necessary to work directly with the *IAppServer* or *IAppServerSOAP* interface, you may have added your own extensions to the remote data module's interface. When you extend the application server's interface, you need a way to call those extensions using the connection created by your connection component. Unless you are using SOAP, you can do this using the *AppServer* property of the connection component. For information about extending the application server's interface, see "Extending the application server's interface" on page 31-16.

*AppServer* is a Variant that represents the application server's interface. If you are not using SOAP, you can call an interface method using *AppServer* by writing a statement such as

```
MyConnection.AppServer.SpecialMethod(x,y);
```

However, this technique provides late (dynamic) binding of the interface call. That is, the *SpecialMethod* procedure call is not bound until runtime when the call is executed. Late binding is very flexible, but by using it you lose many benefits such as code insight and type checking. In addition, late binding is slower than early binding, because the compiler generates additional calls to the server to set up interface calls before they are invoked.

### Using early binding with DCOM

When you are using DCOM as a communications protocol, you can use early binding of *AppServer* calls. Use the **as** operator to cast the *AppServer* variable to the *IAppServer* descendant you created when you created the remote data module. For example:

```
with MyConnection.AppServer as IMyAppServer do
  SpecialMethod(x,y);
```

To use early binding under DCOM, the server's type library must be registered on the client machine. You can use TRegsvr.exe, which ships with Delphi to register the type library.

**Note** See the TRegSvr demo (which provides the source for TRegsvr.exe) for an example of how to register the type library programmatically.

### Using dispatch interfaces with TCP/IP or HTTP

When you are using TCP/IP or HTTP, you can't use true early binding, but because the remote data module uses a dual interface, you can use the application server's dispinterface to improve performance over simple late binding. The dispinterface has the same name as the remote data module's interface, with the string 'Disp' appended. You can assign the *AppServer* property to a variable of this type to obtain the dispinterface. Thus:

```
var
  TempInterface: IMyAppServerDisp;
begin
  TempInterface :=IMyAppServerDisp(IDispatch(MyConnection.AppServer));
ƒ
  TempInterface.SpecialMethod(x,y);
ƒ
end;
```

**Note** To use the dispinterface, you must add the _TLB unit that is generated when you save the type library to the **uses** clause of your client module.

### Calling the interface of a SOAP-based server

If you are using SOAP, you can't use the *AppServer* property. Instead, you must obtain the server's interface by calling the *GetSOAPServer* method. Before you call *GetSOAPServer*, however, you must take the following steps:

- Your client application must include the definition of the application server's interface and register it with the invocation registry. You can add the definition of this interface to your client application by referencing a WSDL document that describes the interface you want to call. For information on importing a WSDL document that describes the server interface, see "Importing WSDL documents" on page 38-20. When you import the interface definition, the WSDL importer automatically adds code to register it with the invocation registry. For more information about interfaces and the invocation registry, see "Understanding invokable interfaces" on page 38-2.
- The *TSOAPConnection* component must have its *UseSOAPAdapter* property set to *True*. This means that the server must support the *IAppServerSOAP* interface. If the application server is built using Delphi 6 or Kylix 1, it does not support *IAppServerSOAP* and you must use a separate *THTTPRio* component instead. For details on how to call an interface using a *THTTPRio* component, see "Calling invokable interfaces" on page 38-20.
- You must set the *SOAPServerIID* property of the SOAP connection component to the GUID of the server interface. You must set this property before your application connects to the server, because it tells the *TSOAPConnection* component what interface to fetch from the server.

Assuming the previous three conditions are met, you can fetch the server interface as follows:

```
with MyConnection.GetSOAPServer as IMyAppServer do
  SpecialMethod(x,y);
```

## Connecting to an application server that uses multiple data modules

If a COM-based application server uses a main "parent" remote data module and several child remote data modules, as described in "Using multiple remote data modules" on page 31-21, then you need a separate connection component for every remote data module on the application server. Each connection component represents the connection to a single remote data module.

While it is possible to have your client application form independent connections to each remote data module on the application server, it is more efficient to use a single connection to the application server that is shared by all the connection components. That is, you add a single connection component that connects to the "main" remote data module on the application server, and then, for each "child" remote data module, add an additional component that shares the connection to the main remote data module.

1 For the connection to the main remote data module, add and set up a connection component as described in "Connecting to the application server" on page 31-23. The only limitation is that you can't use a SOAP connection.

2 For each child remote data module, use a *TSharedConnection* component.

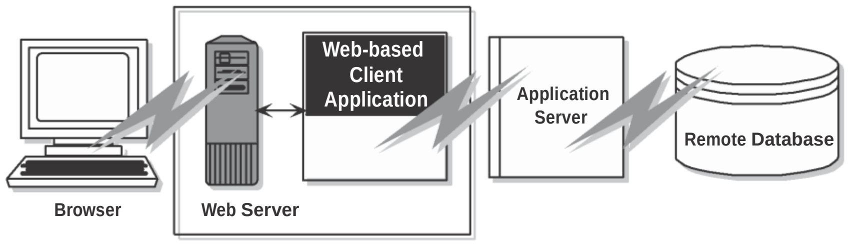
   - Set its *ParentConnection* property to the connection component you added in step 1. The *TSharedConnection* component shares the connection that this main connection establishes.
   - Set its *ChildName* property to the name of the property on the main remote data module's interface that exposes the interface of the desired child remote data module.

When you assign the *TSharedConnection* component placed in step 2 as the value of a client dataset's *RemoteServer* property, it works as if you were using an entirely independent connection to the child remote data module. However, the *TSharedConnection* component uses the connection established by the component you placed in step 1.

## Writing Web-based client applications

If you want to create Web-based clients for your multi-tiered database application, you must replace the client tier with a special Web application that acts simultaneously as a client to an application server and as a Web server application that is installed with a Web server on the same machine. This architecture is illustrated in Figure 31.1.

**Figure 31.1** Web-based multi-tiered database application



There are two approaches that you can take to build the Web application:

- You can combine the multi-tiered database architecture with an ActiveX form to distribute the client application as an ActiveX control. This allows any browser that supports ActiveX to run your client application as an in-process server.
- You can use XML data packets to build an InternetExpress application. This allows browsers that supports javascript to interact with your client application through html pages.

These two approaches are very different. Which one you choose depends on the following considerations:

- Each approach relies on a different technology (ActiveX vs. javascript and XML). Consider what systems your end users will use. The first approach requires a browser to support ActiveX (which limits clients to a Windows platform). The second approach requires a browser to support javascript and the DHTML capabilities introduced by Netscape 4 and Internet Explorer 4.
- ActiveX controls must be downloaded to the browser to act as an in-process server. As a result, the clients using an ActiveX approach require much more memory than the clients of an HTML-based application.
- The InternetExpress approach can be integrated with other HTML pages. An ActiveX client must run in a separate window.
- The InternetExpress approach uses standard HTTP, thereby avoiding any firewall issues that confront an ActiveX application.
- The ActiveX approach provides greater flexibility in how you program your application. You are not limited by the capabilities of the javascript libraries. The client datasets used in the ActiveX approach surface more features (such as filters, ranges, aggregation, optional parameters, delayed fetching of BLOBs or nested details, and so on) than the XML brokers used in the InternetExpress approach.

**Caution** Your Web client application may look and act differently when viewed from different browsers. Test your application with the browsers you expect your end-users to use.

## Distributing a client application as an ActiveX control

The multi-tiered database architecture can be combined with ActiveX features to distribute a client application as an ActiveX control.

When you distribute your client application as an ActiveX control, create the application server as you would for any other multi-tiered application. For details on creating the application server, see “Creating the application server” on page 31-12.

When creating the client application, you must use an Active Form as the basis instead of an ordinary form. See “Creating an Active Form for the client application” for details.

Once you have built and deployed your client application, it can be accessed from any ActiveX-enabled Web browser on another machine. For a Web browser to successfully launch your client application, the Web server must be running on the machine that has the client application.

If the client application uses DCOM to communicate between the client application and the application server, the machine with the Web browser must be enabled to work with DCOM. If the machine with the Web browser is a Windows 95 machine, it must have installed DCOM95, which is available from Microsoft.

### Creating an Active Form for the client application

1. Because the client application will be deployed as an ActiveX control, you must have a Web server that runs on the same system as the client application. You can use a ready-made server such as Microsoft's Personal Web server or you can write your own using the socket components described in Chapter 39, "Working with sockets."
2. Create the client application following the steps described in "Creating the client application" on page 31-22, except start by choosing File | New | ActiveX | Active Form, rather than beginning an ordinary client project.
3. If your client application uses a data module, add a call to explicitly create the data module in the active form initialization.
4. When your client application is finished, compile the project, and select Project | Web Deployment Options. In the Web Deployment Options dialog, you must do the following:
   a. On the Project page, specify the Target directory, the URL for the target directory, and the HTML directory. Typically, the Target directory and the HTML directory will be the same as the projects directory for your Web Server. The target URL is typically the name of the server machine.
   b. On the Additional Files page, include midas.dll with your client application.
5. Finally, select Project | WebDeploy to deploy the client application as an active form.

Any Web browser that can run Active forms can run your client application by specifying the .HTM file that was created when you deployed the client application. This .HTM file has the same name as your client application project, and appears in the directory specified as the Target directory.

## Building Web applications using InternetExpress

A client application can request that the application server provide data packets that are coded in XML instead of OleVariants. By combining XML-coded data packets, special javascript libraries of database functions, and the Web server application support, you can create thin client applications that can be accessed using a Web browser that supports javascript. This combination of features is called InternetExpress.

Before building an InternetExpress application, you should understand the Web server application architecture. This is described in Chapter 33, "Creating Internet server applications."

An InternetExpress application extends the basic Web server application architecture to act as the client of an application server. InternetExpress applications generate HTML pages that contain a mixture of HTML, XML, and javascript. The HTML governs the layout and appearance of the pages seen by end users in their browsers. The XML encodes the data packets and delta packets that represent database information. The javascript allows the HTML controls to interpret and manipulate the data in these XML data packets on the client machine.

If the InternetExpress application uses DCOM to connect to the application server, you must take additional steps to ensure that the application server grants access and launch permissions to its clients. See "Granting permission to access and launch the application server" on page 31-36 for details.

**Tip** You can create an InternetExpress application to provide Web browsers with "live" data even if you do not have an application server. Simply add the provider and its dataset to the Web module.

## Building an InternetExpress application

The following steps describe one way to build a Web application using InternetExpress. The result is an application that creates HTML pages that let users interact with the data from an application server via a javascript-enabled Web browser. You can also build an InternetExpress application using the Site Express architecture by using the InternetExpress page producer (*TInetXPageProducer*).

1. Choose File|New|Other to display the New Items dialog box, and on the New page select Web Server application. This process is described in "Creating Web server applications with Web Broker" on page 34-1.
2. From the DataSnap page of the Component palette, add a connection component to the Web Module that appears when you create a new Web server application. The type of connection component you add depends on the communication protocol you want to use. See "Choosing a connection protocol" on page 31-9 for details.
3. Set properties on your connection component to specify the application server with which it should establish a connection. To learn more about setting up the connection component, see "Connecting to the application server" on page 31-23.
4. Instead of a client dataset, add an XML broker from the InternetExpress page of the Component palette to the Web module. Like *TClientDataSet*, *TXMLBroker* represents the data from a provider on the application server and interacts with the application server through an *IAppServer* interface. However, unlike client datasets, XML brokers request data packets as XML instead of as OleVariants and interact with InternetExpress components instead of data controls.
5. Set the *RemoteServer* property of the XML broker to point to the connection component you added in step 2. Set the *ProviderName* property to indicate the provider on the application server that provides data and applies updates. For more information about setting up the XML broker, see "Using an XML broker" on page 31-36.
6. Add an InternetExpress page producer (*TInetXPageProducer*) to the Web module for each separate page that users will see in their browsers. For each page producer, you must set the *IncludePathURL* property to indicate where it can find the javascript libraries that augment its generated HTML controls with data management capabilities.

7 Right-click a Web page and choose Action Editor to display the Action editor. Add action items for every message you want to handle from browsers. Associate the page producers you added in step 6 with these actions by setting their *Producer* property or writing code in an *OnAction* event handler. For more information on adding action items using the Action editor, see “Adding actions to the dispatcher” on page 34-5.

8 Double-click each Web page to display the Web Page editor. (You can also display this editor by clicking the ellipsis button in the Object Inspector next to the *WebPageItems* property.) In this editor you can add Web Items to design the pages that users see in their browsers. For more information about designing Web pages for your InternetExpress application, see “Creating Web pages with an InternetExpress page producer” on page 31-39.

9 Build your Web application. Once you install this application with your Web server, browsers can call it by specifying the name of the application as the script name portion of the URL and the name of the Web Page component as the pathinfo portion.

## Using the javascript libraries

The HTML pages generated by the InternetExpress components and the Web items they contain make use of several javascript libraries that ship in the source/webmidas directory:

**Table 31.3** Javascript libraries

| Library | Description |
|---|---|
| xmldom.js | This library is a DOM-compatible XML parser written in javascript. It allows parsers that do not support XML to use XML data packets. Note that this does not include support for XML Islands, which are supported by IE5 and later. |
| xmldb.js | This library defines data access classes that manage XML data packets and XML delta packets. |
| xmldisp.js | This library defines classes that associate the data access classes in xmldb with HTML controls in the HTML page. |
| xmlerrdisp.js | This library defines classes that can be used when reconciling update errors. These classes are not used by any of the built-in InternetExpress components, but are useful when writing a Reconcile producer. |
| xmlshow.js | This library includes functions to display formatted XML data packets and XML delta packets. This library is not used by any of the InternetExpress components, but is useful when debugging. |

Once you have installed these libraries, you must set the *IncludePathURL* property of all InternetExpress page producers to indicate where they can be found.

It is possible to write your own HTML pages using the javascript classes provided in these libraries instead of using Web items to generate your Web pages. However, you must ensure that your code does not do anything illegal, as these classes include minimal error checking (so as to minimize the size of the generated Web pages).

### Granting permission to access and launch the application server

Requests from the InternetExpress application appear to the application server as originating from a guest account with the name IUSR_computername, where computername is the name of the system running the Web application. By default, this account does not have access or launch permission for the application server. If you try to use the Web application without granting these permissions, when the Web browser tries to load the requested page it times out with `EOLE_ACCESS_ERROR`.

**Note** Because the application server runs under this guest account, it can't be shut down by other accounts.

To grant the Web application access and launch permissions, run DCOMCnfg.exe, which is located in the System32 directory of the machine that runs the application server. The following steps describe how to configure your application server:

1. When you run DCOMCnfg, select your application server in the list of applications on the Applications page.
2. Click the Properties button. When the dialog changes, select the Security page.
3. Select Use Custom Access Permissions, and press the Edit button. Add the name IUSR_computername to the list of accounts with access permission, where computername is the name of the machine that runs the Web application.
4. Select Use Custom Launch Permissions, and press the Edit button. Add IUSR_computername to this list as well.
5. Click the Apply button.

## Using an XML broker

The XML broker serves two major functions:

- It fetches XML data packets from the application server and makes them available to the Web Items that generate HTML for the InternetExpress application.
- It receives updates in the form of XML delta packets from browsers and applies them to the application server.

### Fetching XML data packets

Before the XML broker can supply XML data packets to the components that generate HTML pages, it must fetch them from the application server. To do this, it uses the *IAppServer* interface, which it acquires from a connection component.

**Note** Even when using SOAP, where the application server supports *IAppServerSOAP*, the XML broker uses *IAppServer* because the connection component acts as an adapter between the two interfaces.

You must set the following properties so that the XML producer can use the *IAppServer* interface:

- Set the *RemoteServer* property to the connection component that establishes the connection to the application server and gets its *IAppServer* interface. At design time, you can select this value from a drop-down list in the object inspector.
- Set the *ProviderName* property to the name of the provider component on the application server that represents the dataset for which you want XML data packets. This provider both supplies XML data packets and applies updates from XML delta packets. At design time, if the *RemoteServer* property is set and the connection component has an active connection, the Object Inspector displays a list of available providers. (If you are using a DCOM connection the application server must also be registered on the client machine).

Two properties let you indicate what you want to include in data packets:

- You can limit the number of records that are added to the data packet by setting the *MaxRecords* property. This is especially important for large datasets because InternetExpress applications send the entire data packet to client Web browsers. If the data packet is too large, the download time can become prohibitively long.
- If the provider on the application server represents a query or stored procedure, you may want to provide parameter values before obtaining an XML data packet. You can supply these parameter values using the *Params* property.

The components that generate HTML and javascript for the InternetExpress application automatically use the XML broker's XML data packet once you set their *XMLBroker* property. To obtain the XML data packet directly in code, use the *RequestRecords* method.

**Note** When the XML broker supplies a data packet to another component (or when you call *RequestRecords*), it receives an *OnRequestRecords* event. You can use this event to supply your own XML string instead of the data packet from the application server. For example, you could fetch the XML data packet from the application server using *GetXMLRecords* and then edit it before supplying it to the emerging Web page.

### Applying updates from XML delta packets

When you add the XML broker to the Web module (or data module containing a *TWebDispatcher*), it automatically registers itself with the Web dispatcher as an auto-dispatching object. This means that, unlike other components, you do not need to create an action item for the XML broker in order for it to respond to update messages from a Web browser. These messages contain XML delta packets that should be applied to the application server. Typically, they originate from a button that you create on one of the HTML pages produced by the Web client application.

So that the dispatcher can recognize messages for the XML broker, you must describe them using the *WebDispatch* property. Set the *PathInfo* property to the path portion of the URL to which messages for the XML broker are sent. Set *MethodType* to the value of the method header of update messages addressed to that URL (typically *mtPost*). If you want to respond to all messages with the specified path, set *MethodType* to *mtAny*. If you don't want the XML broker to respond directly to update messages (for example, if you want to handle them explicitly using an action item), set the *Enabled* property to *False*. For more information on how the Web dispatcher determines which component handles messages from the Web browser, see "Dispatching request messages" on page 34-5.

When the dispatcher passes an update message on to the XML broker, it passes the updates on to the application server and, if there are update errors, receives an XML delta packet describing all update errors. Finally, it sends a response message back to the browser, which either redirects the browser to the same page that generated the XML delta packet or sends it some new content.

A number of events allow you to insert custom processing at all steps of this update process:

1 When the dispatcher first passes the update message to the XML broker, it receives a *BeforeDispatch* event, where you can preprocess the request or even handle it entirely. This event allows the XML broker to handle messages other than update messages.
2 If the *BeforeDispatch* event handler does not handle the message, the XML broker receives an *OnRequestUpdate* event, where you can apply the updates yourself rather than using the default processing.
3 If the *OnRequestUpdate* event handler does not handle the request, the XML broker applies the updates and receives a delta packet containing any update errors.
4 If there are no update errors, the XML broker receives an *OnGetResponse* event, where you can create a response message that indicates the updates were successfully applied or sends refreshed data to the browser. If the *OnGetResponse* event handler does not complete the response (does not set the *Handled* parameter to *True*), the XML broker sends a response that redirects the browser back to the document that generated the delta packet.
5 If there are update errors, the XML broker receives an *OnGetErrorResponse* event instead. You can use this event to try to resolve update errors or to generate a Web page that describes them to the end user. If the *OnGetErrorResponse* event handler does not complete the response (does not set the *Handled* parameter to *True*), the XML broker calls on a special content producer called the *ReconcileProducer* to generate the content of the response message.
6 Finally, the XML broker receives an *AfterDispatch* event, where you can perform any final actions before sending a response back to the Web browser.

## Creating Web pages with an InternetExpress page producer

Each InternetExpress page producer generates an HTML document that appears in the browsers of your application's clients. If your application includes several separate Web documents, use a separate page producer for each of them.

The InternetExpress page producer (*TInetXPageProducer*) is a special page producer component. As with other page producers, you can assign it as the *Producer* property of an action item or call it explicitly from an *OnAction* event handler. For more information about using content producers with action items, see "Responding to request messages with action items" on page 34-8. For more information about page producers, see "Using page producer components" on page 34-14.

The InternetExpress page producer has a default template as the value of its *HTMLDoc* property. This template contains a set of HTML-transparent tags that the InternetExpress page producer uses to assemble an HTML document (with embedded javascript and XML) including content produced by other components. Before it can translate all of the HTML-transparent tags and assemble this document, you must indicate the location of the javascript libraries used for the embedded javascript on the page. This location is specified by setting the *IncludePathURL* property.

You can specify the components that generate parts of the Web page using the Web page editor. Display the Web page editor by double-clicking the Web page component or clicking the ellipsis button next to the *WebPageItems* property in the Object Inspector.

The components you add in the Web page editor generate the HTML that replaces one of the HTML-transparent tags in the InternetExpress page producer's default template. These components become the value of the *WebPageItems* property. After adding the components in the order you want them, you can customize the template to add your own HTML or change the default tags.

### Using the Web page editor

The Web page editor lets you add Web items to your InternetExpress page producer and view the resulting HTML page. Display the Web page editor by double-clicking on a InternetExpress page producer component.

**Note** You must have Internet Explorer 4 or better installed to use the Web page editor.

The top of the Web page editor displays the Web items that generate the HTML document. These Web items are nested, where each type of Web item assembles the HTML generated by its subitems. Different types of items can contain different subitems. On the left, a tree view displays all of the Web items, indicating how they are nested. On the right, you can see the Web items included by the currently selected item. When you select a component in the top of the Web page editor, you can set its properties using the Object Inspector.

Click the New Item button to add a subitem to the currently selected item. The Add Web Component dialog lists only those items that can be added to the currently selected item.

The InternetExpress page producer can contain one of two types of item, each of which generates an HTML form:

- *TDataForm*, which generates an HTML form for displaying data and the controls that manipulate that data or submit updates.

  Items you add to *TDataForm* display data in a multi-record grid (*TDataGrid*) or in a set of controls each of which represents a single field from a single record (*TFieldGroup*). In addition, you can add a set of buttons to navigate through data or post updates (*TDataNavigator*), or a button to apply updates back to the Web client (*TApplyUpdatesButton*). Each of these items contains subitems to represent individual fields or buttons. Finally, as with most Web items, you can add a layout grid (*TLayoutGroup*), that lets you customize the layout of any items it contains.
- *TQueryForm*, which generates an HTML form for displaying or reading application-defined values. For example, you can use this form for displaying and submitting parameter values.

  Items you add to *TQueryForm* display application-defined values(*TQueryFieldGroup*) or a set of buttons to submit or reset those values (*TQueryButtons*). Each of these items contains subitems to represent individual values or buttons. You can also add a layout grid to a query form, just as you can to a data form.

The bottom of the Web page editor displays the generated HTML code and lets you see what it looks like in a browser (Internet Explorer).

## Setting Web item properties

The Web items that you add using the Web page editor are specialized components that generate HTML. Each Web item class is designed to produce a specific control or section of the final HTML document, but a common set of properties influences the appearance of the final HTML.

When a Web item represents information from the XML data packet (for example, when it generates a set of field or parameter display controls or a button that manipulates the data), the *XMLBroker* property associates the Web item with the XML broker that manages the data packet. You can further specify a dataset that is contained in a dataset field of that data packet using the *XMLDataSetField* property. If the Web item represents a specific field or parameter value, the Web item has a *FieldName* or *ParamName* property.

You can apply a style attribute to any Web item, thereby influencing the overall appearance of all the HTML it generates. Styles and style sheets are part of the HTML 4 standard. They allow an HTML document to define a set of display attributes that apply to a tag and everything in its scope. Web items offer a flexible selection of ways to use them:

- The simplest way to use styles is to define a style attribute directly on the Web item. To do this, use the *Style* property. The value of *Style* is simply the attribute definition portion of a standard HTML style definition, such as `color: red`.
- You can also define a style sheet that defines a set of style definitions. Each definition includes a style selector (the name of a tag to which the style always applies or a user-defined style name) and the attribute definition in curly braces:

  ```
  H2 B {color: red}
  .MyStyle {font-family: arial; font-weight: bold; font-size: 18px }
  ```

  The entire set of definitions is maintained by the InternetExpress page producer as its *Styles* property. Each Web item can then reference the styles with user-defined names by setting its *StyleRule* property.
- If you are sharing a style sheet with other applications, you can supply the style definitions as the value of the InternetExpress page producer's *StylesFile* property instead of the *Styles* property. Individual Web items still reference styles using the *StyleRule* property.

Another common property of Web items is the *Custom* property. *Custom* includes a set of options that you add to the generated HTML tag. HTML defines a different set of options for each type of tag. The VCL reference for the *Custom* property of most Web items gives an example of possible options. For more information on possible options, use an HTML reference.

## Customizing the InternetExpress page producer template

The template of an InternetExpress page producer is an HTML document with extra embedded tags that your application translates dynamically. Initially, the page producer generates a default template as the value of the *HTMLDoc* property. This default template has the form

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<#INCLUDES> <#STYLES> <#WARNINGS> <#FORMS> <#SCRIPT>
</BODY>
</HTML>
```

The HTML-transparent tags in the default template are translated as follows:

**<#INCLUDES>** generates the statements that include the javascript libraries. These statements have the form

```
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldom.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldb.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmlbind.js"> </
    SCRIPT>
```

**<#STYLES>** generates the statements that defines a style sheet from definitions listed in the *Styles* or *StylesFile* property of the InternetExpress page producer.

**<#WARNINGS>** generates nothing at runtime. At design time, it adds warning messages for problems detected while generating the HTML document. You can see these messages in the Web page editor.

**<#FORMS>** generates the HTML produced by the components that you add in the Web page editor. The HTML from each component is generated in the order it appears in *WebPageItems*.

**<#SCRIPT>** generates a block of javascript declarations that are used in the HTML generated by the components added in the Web page editor.

You can replace the default template by changing the value of *HTMLDoc* or setting the *HTMLFile* property. The customized HTML template can include any of the HTML-transparent tags that make up the default template. The InternetExpress page producer automatically translates these tags when you call the *Content* method. In addition, The InternetExpress page producer automatically translates three additional tags:

**<#BODYELEMENTS>** is replaced by the same HTML as results from the 5 tags in the default template. It is useful when generating a template in an HTML editor when you want to use the default layout but add additional elements using the editor.

**<#COMPONENT Name=WebComponentName>** is replaced by the HTML that the component named *WebComponentName* generates. This component can be one of the components added in the Web page editor, or it can be any component that supports the *IWebContent* interface and has the same Owner as the InternetExpress page producer.

**<#DATAPACKET XMLBroker=BrokerName>** is replaced with the XML data packet obtained from the XML broker specified by *BrokerName*. When, in the Web page editor, you see the HTML that the InternetExpress page producer generates, you see this tag instead of the actual XML data packet.

In addition, the customized template can include any other HTML-transparent tags that you define. When the InternetExpress page producer encounters a tag that is not one of the seven types it translates automatically, it generates an *OnHTMLTag* event, where you can write code to perform your own translations. For more information about HTML templates in general, see “HTML templates” on page 34-14.

**Tip** The components that appear in the Web page editor generate static code. That is, unless the application server changes the metadata that appears in data packets, the HTML is always the same, no matter when it is generated. You can avoid the overhead of generating this code dynamically at runtime in response to every request message by copying the generated HTML in the Web page editor and using it as a template. Because the Web page editor displays a <#DATAPACKET> tag instead of the actual XML, using this as a template still allows your application to fetch data packets from the application server dynamically.

Chapter

32

# Using XML in database applications

In addition to the support for connecting to database servers, Delphi lets you work with XML documents as if they were database servers. XML (Extensible Markup Language) is a markup language for describing structured data. XML documents provide a standard, transportable format for data that is used in Web applications, business-to-business communication, and so on. For information on Delphi's support for working directly with XML documents, see Chapter 37, "Working with XML documents."

Support for working with XML documents in database applications is based on a set of components that can convert data packets (the *Data* property of a client dataset) into XML documents and convert XML documents into data packets. To use these components, you must first define the transformation between the XML document and the data packet. Once you have defined the transformation, you can use special components to

- convert XML documents into data packets.
- provide data from and resolve updates to an XML document.
- use an XML document as the client of a provider.

## Defining transformations

Before you can convert between data packets and XML documents, you must define the relationship between the metadata in a data packet and the nodes of the corresponding XML document. A description of this relationship is stored in a special XML document called a transformation.

Each transformation file contains two things: the mapping between the nodes in an XML schema and the fields in a data packet, and a skeletal XML document that represents the structure for the results of the transformation. A transformation is a one-way mapping: from an XML schema or document to a data packet or from the metadata in a data packet to an XML schema. Often, you create transformation files in pairs: one that maps from XML to data packet, and one that maps from data packet to XML.

In order to create the transformation files for a mapping, use the XMLMapper utility that ships in the bin directory.

## Mapping between XML nodes and data packet fields

XML provides a text-based way to store or describe structured data. Datasets provide another way to store and describe structured data. To convert an XML document into a dataset, therefore, you must identify the correspondences between the nodes in an XML document and the fields in a dataset.

Consider, for example, an XML document that represents a set of email messages. It might look like the following (containing a single message):

```
<?xml version="1.0" standalone="yes" ?>
<email>
   <head>
      <from>
         <name>Dave Boss</name>
         <address>dboss@MyCo.com</address>
      </from>
      <to>
         <name>Joe Engineer</name>
         <address>jengineer@MyCo.com</address>
      </to>
      <cc>
         <name>Robin Smith/name>
         <address>rsmith@MyCo.com</address>
      </cc>
      <cc>
         <name>Leonard Devon</name>
         <address>ldevon@MyCo.com</address>
      </cc>
   </head>
   <body>
      <subject>XML components</subject>
      <content>
        Joe,
        Attached is the specification for the XML component support in Delphi.
        This looks like a good solution to our buisness-to-buisness application!
        Also attached, please find the project schedule. Do you think its reasonable?
           Dave.
      </content>
      <attachment attachfile="XMLSpec.txt"/>
      <attachment attachfile="Schedule.txt"/>
   </body>
</email>
```

One natural mapping between this document and a dataset would map each e-mail message to a single record. The record would have fields for the sender's name and address. Because an e-mail message can have multiple recipients, the recipient (<to> would map to a nested dataset. Similarly, the cc list maps to a nested dataset. The subject line would map to a string field while the message itself (<content>) would probably be a memo field. The names of attachment files would map to a nested dataset because one message can have several attachments. Thus, the e-mail above would map to a dataset something like the following:

| SenderName | SenderAddress | To | CC | Subject | Content | Attach |
|---|---|---|---|---|---|---|
| Dave Boss | dboss@MyCo.Com | (DataSet) | (DataSet) | XML components | (MEMO) | (DataSet) |

where the nested dataset in the "To" field is

| Name | Address |
|---|---|
| Joe Engineer | jengineer@MyCo.Com |

the nested dataset in the "CC" field is

| Name | Address |
|---|---|
| Robin Smith | rsmith@MyCo.Com |
| Leonard Devon | ldevon@MyCo.Com |

and the nested dataset in the "Attach" field is

| Attachfile |
|---|
| XMLSpec.txt |
| Schedule.txt |

Defining such a mapping involves identifying those nodes of the XML document that can be repeated and mapping them to nested datasets. Tagged elements that have values and appear only once (such as <content>...</content>) map to fields whose datatype reflects the type of data that can appear as the value. Attributes of a tag (such as the AttachFile attribute of the attachment tag) also map to fields.

Note that not all tags in the XML document appear in the corresponding dataset. For example, the <head>...<head/> element has no corresponding element in the resulting dataset. Typically, only elements that have values, elements that can be repeated, or the attributes of a tag map to the fields (including nested dataset fields) of a dataset. The exception to this rule is when a parent node in the XML document maps to a field whose value is built up from the values of the child nodes. For example, an XML document might contain a set of tags such as

```
<FullName>
    <Title> Mr. </Title>
    <FirstName> John </FirstName>
    <LastName> Smith </LastName>
</FullName>
```

which could map to a single dataset field with the value

```
Mr. John Smith
```

## Using XMLMapper

The XML mapper utility, xmlmapper.exe, lets you define mappings in three ways:

- From an existing XML schema (or document) to a client dataset that you define. This is useful when you want to create a database application to work with data for which you already have an XML schema.
- From an existing data packet to a new XML schema you define. This is useful when you want to expose existing database information in XML, for example to create a new business-to-business communication system.
- Between an existing XML schema and an existing data packet. This is useful when you have an XML schema and a database that both describe the same information and you want to make them work together.

Once you define the mapping, you can generate the transformation files that are used to convert XML documents to data packets and to convert data packets to XML documents. Note that only the transformation file is directional: a single mapping can be used to generate both the transformation from XML to data packet and from data packet to XML.

**Note** XML mapper relies on two .DLLs (midas.dll and msxml.dll) to work correctly. Be sure that you have both of these .DLLs installed before you try to use xmlmapper.exe. In addition, msxml.dll must be registered as a COM server. You can register it using Regsvr32.exe.

### Loading an XML schema or data packet

Before you can define a mapping and generate a transformation file, you must first load descriptions of the XML document and the data packet between which you are mapping.

You can load an XML document or schema by choosing File|Open and selecting the document or schema in the resulting dialog.

You can load a data packet by choosing File|Open and selecting a data packet file in the resulting dialog. (The data packet is simply the file generated when you call a client dataset's *SaveToFile* method.) If you have not saved the data packet to disk, you can fetch the data packet directly from the application server of a multi-tiered application by right-clicking in the Datapacket pane and choosing Connect To Remote Server.

You can load only an XML document or schema, only a data packet, or you can load both. If you load only one side of the mapping, XML mapper can generate a natural mapping for the other side.

## Defining mappings

The mapping between an XML document and a data packet need not include all of the fields in the data packet or all of the tagged elements in the XML document. Therefore, you must first specify those elements that are mapped. To specify these elements, first select the Mapping page in the central pane of the dialog.

To specify the elements of an XML document or schema that are mapped to fields in a data packet, select the Sample or Structure tab of the XML document pane and double-click on the nodes for elements that map to data packet fields.

To specify the fields of the data packet that are mapped to tagged elements or attributes in the XML document, double-click on the nodes for those fields in the Datapacket pane.

If you have only loaded one side of the mapping (the XML document or the data packet), you can generate the other side after you have selected the nodes that are mapped.

- If you are generating a data packet from an XML document, you first define attributes for the selected nodes that determine the types of fields to which they correspond in the data packet. In the center pane, select the Node Repository page. Select each node that participates in the mapping and indicate the attributes of the corresponding field. If the mapping is not straightforward (for example, a node with subnodes that corresponds to a field whose value is built from those subnodes), check the User Defined Translation check box. You will need to write an event handler later to perform the transformation on user defined nodes.

  Once you have specified the way nodes are to be mapped, choose Create | Datapacket from XML. The corresponding data packet is automatically generated and displayed in the Datapacket pane.
- If you are generating an XML document from a data packet, choose Create | XML from Datapacket. A dialog appears where you can specify the names of the tags and attributes in the XML document that correspond to fields, records, and datasets in the data packet. For field values, the way you name them indicates whether they map to a tagged element with a value or to an attribute. Names that begin with an @ symbol map to attributes of the tag that corresponds to the record, while names that do not begin with an @ symbol map to tagged elements that have values and that are nested within the element for the record.
- If you have loaded both an XML document and a data packet (client dataset file), be sure you select corresponding nodes in the same order. The corresponding nodes should appear next to each other in the table at the top of the Mapping page.

Once you have loaded or generated both the XML document and the data packet and selected the nodes that appear in the mapping, the table at the top of the Mapping page should reflect the mapping you have defined.

### Generating transformation files

To generate a transformation file, use the following steps:

1. First select the radio button that indicates what the transformation creates:
   - Choose the Datapacket to XML button if the mapping goes from data packet to XML document.
   - Choose the XML to Datapacket button if the mapping goes from XML document to data packet.
2. If you are generating a data packet, you will also want to use the radio buttons in the Create Datapacket As section. These buttons let you specify how the data packet will be used: as a dataset, as a delta packet for applying updates, or as the parameters to supply to a provider before fetching data.
3. Click Create and Test Transformation to generate an in-memory version of the transformation. XML mapper displays the XML document that would be generated for the data packet in the Datapacket pane or the data packet that would be generated for the XML document in the XML Document pane.
4. Finally, choose File|Save|Transformation to save the transformation file. The transformation file is a special XML file (with the .xtr extension) that describes the transformation you have defined.

## Converting XML documents into data packets

Once you have created a transformation file that indicates how to transform an XML document into a data packet, you can create data packets for any XML document that conforms to the schema used in the transformation. These data packets can then be assigned to a client dataset and saved to a file so that they form the basis of a file-based database application.

The *TXMLTransform* component transforms an XML document into a data packet according to the mapping in a transformation file.

**Note** You can also use *TXMLTransform* to convert a data packet that appears in XML format into an arbitrary XML document.

### Specifying the source XML document

There are three ways to specify the source XML document:

- If the source document is an .xml file on disk, you can use the *SourceXmlFile* property.
- If the source document is an in-memory string of XML, you can use the *SourceXml* property.
- If you have an IDOMDocument interface for the source document, you can use the *SourceXmlDocument* property.

*TXMLTransform* checks these properties in the order listed above. That is, it first checks for a file name in the *SourceXmlFile* property. Only if *SourceXmlFile* is an empty string does it check the *SourceXml* property. Only if *SourceXml* is an empty string does it then check the *SourceXmlDocument* property.

## Specifying the transformation

There are two ways to specify the transformation that converts the XML document into a data packet:

- Set the *TransformationFile* property to indicate a transformation file that was created using xmlmapper.exe.
- Set the *TransformationDocument* property if you have an *IDOMDocument* interface for the transformation.

*TXMLTransform* checks these properties in the order listed above. That is, it first checks for a file name in the *TransformationFile* property. Only if *TransformationFile* is an empty string does it check the *TransformationDocument* property.

## Obtaining the resulting data packet

To cause *TXMLTransform* to perform its transformation and generate a data packet, you need only read the *Data* property. For example, the following code uses an XML document and transformation file to generate a data packet, which is then assigned to a client dataset:

```
XMLTransform1.SourceXMLFile := 'CustomerDocument.xml';
XMLTransform1.TransformationFile := 'CustXMLToCustTable.xtr';
ClientDataSet1.XMLData := XMLTransform1.Data;
```

## Converting user-defined nodes

When you define a transformation using xmlmapper.exe, you can specify that some of the nodes in the XML document are “user-defined.” User-defined nodes are nodes for which you want to provide the transformation in code rather than relying on a straightforward node-value-to-field-value translation.

You can provide the code to translate user-defined nodes using the *OnTranslate* event. The *OnTranslate* event handler is called every time the *TXMLTransform* component encounters a user-defined node in the XML document. In the OnTranslate event handler, you can read the source document and specify the resulting value for the field in the data packet.

For example, the following *OnTranslate* event handler converts a node in the XML document with the following form

```
<FullName>
   <Title> </Title>
   <FirstName> </FirstName>
   <LastName> </LastName>
</FullName>
```

into a single field value:

```
procedure TForm1.XMLTransform1Translate(Sender: TObject; Id: String; SrcNode: IDOMNode;
  var Value: String; DestNode: IDOMNode);
var
  CurNode: IDOMNode;
begin
  if Id = 'FullName' then
  begin
    Value = '';
    if SrcNode.hasChildNodes then
    begin
      CurNode := SrcNode.firstChild;
      Value := Value + CurNode.nodeValue;
      while CurNode <> SrcNode.lastChild do
      begin
        CurNode := CurNode.nextSibling;
        Value := Value + ' ';
        Value := Value + CurNode.nodeValue;
      end;
    end;
  end;
end;
```

## Using an XML document as the source for a provider

The *TXMLTransformProvider* component lets you use an XML document as if it were a database table. *TXMLTransformProvider* packages the data from an XML document and applies updates from clients back to that XML document. It appears to clients such as client datasets or XML brokers like any other provider component. For information on provider components, see Chapter 30, "Using provider components." For information on using provider components with client datasets, see "Using a client dataset with a provider" on page 29-24.

You can specify the XML document from which the XML provider provides data and to which it applies updates using the *XMLDataFile* property.

*TXMLTransformProvider* components use internal *TXMLTransform* components to translate between data packets and the source XML document: one to translate the XML document into data packets, and one to translate data packets back into the XML format of the source document after applying updates. These two *TXMLTransform* components can be accessed using the *TransformRead* and *TransformWrite* properties, respectively.

When using *TXMLTransformProvider*, you must specify the transformations that these two *TXMLTransform* components use to translate between data packets and the source XML document. You do this by setting the *TXMLTransform* component's *TransformationFile* or *TransformationDocument* property, just as when using a stand-alone *TXMLTransform* component.

In addition, if the transformation includes any user-defined nodes, you must supply an *OnTranslate* event handler to the internal *TXMLTransform* components.

You do not need to specify the source document on the *TXMLTransform* components that are the values of *TransformRead* and *TransformWrite*. For *TransformRead*, the source is the file specified by the provider's *XMLDataFile* property (although, if you set *XMLDataFile* to an empty string, you can supply the source document using *TransformRead.XmlSource* or *TransformRead.XmlSourceDocument*). For *TransformWrite*, the source is generated internally by the provider when it applies updates.

## Using an XML document as the client of a provider

The *TXMLTransformClient* component acts as an adapter to let you use an XML document (or set of documents) as the client for an application server (or simply as the client of a dataset to which it connects via a *TDataSetProvider* component). That is, *TXMLTransformClient* lets you publish database data as an XML document and to make use of update requests (insertions or deletions) from an external application that supplies them in the form of XML documents.

To specify the provider from which the *TXMLTransformClient* object fetches data and to which it applies updates, set the *ProviderName* property. As with the *ProviderName* property of a client dataset, *ProviderName* can be the name of a provider on a remote application server or it can be a local provider in the same form or data module as the *TXMLTransformClient* object. For information about providers, see Chapter 30, "Using provider components."

If the provider is on a remote application server, you must use a DataSnap connection component to connect to that application server. Specify the connection component using the *RemoteServer* property. For information on DataSnap connection components, see "Connecting to the application server" on page 31-23.

### Fetching an XML document from a provider

*TXMLTransformClient* uses an internal *TXMLTransform* component to translate data packets from the provider into an XML document. You can access this *TXMLTransform* component as the value of the *TransformGetData* property.

Before you can create an XML document that represents the data from a provider, you must specify the transformation file that *TransformGetData* uses to translate the data packet into the appropriate XML format. You do this by setting the *TXMLTransform* component's *TransformationFile* or *TransformationDocument* property, just as when using a stand-alone *TXMLTransform* component. If that transformation includes any user-defined nodes, you will want to supply *TransformGetData* with an *OnTranslate* event handler as well.

There is no need to specify the source document for *TransformGetData*, *TXMLTransformClient* fetches that from the provider. However, if the provider expects any input parameters, you may want to set them before fetching the data. Use the *SetParams* method to supply these input parameters before you fetch data from the provider. *SetParams* takes two arguments: a string of XML from which to extract parameter values, and the name of a transformation file to translate that XML into a data packet. *SetParams* uses the transformation file to convert the string of XML into a data packet, and then extracts the parameter values from that data packet.

**Note** You can override either of these arguments if you want to specify the parameter document or transformation in another way. Simply set one of the properties on *TransformSetParams* property to indicate the document that contains the parameters or the transformation to use when converting them, and then set the argument you want to override to an empty string when you call *SetParams*. For details on the properties you can use, see "Converting XML documents into data packets" on page 32-6.

Once you have configured *TransformGetData* and supplied any input parameters, you can call the *GetDataAsXml* method to fetch the XML. *GetDataAsXml* sends the current parameter values to the provider, fetches a data packet, converts it into an XML document, and returns that document as a string. You can save this string to a file:

```
var
  XMLDoc: TFileStream;
  XML: string;
begin
  XMLTransformClient1.ProviderName := 'Provider1';
  XMLTransformClient1.TransformGetData.TransformationFile := 'CustTableToCustXML.xtr';
  XMLTransformClient1.TransFormSetParams.SourceXmlFile := 'InputParams.xml';
  XMLTransformClient1.SetParams('', 'InputParamsToDP.xtr');
  XML := XMLTransformClient1.GetDataAsXml;
  XMLDoc := TFileStream.Create('Customers.xml', fmCreate or fmOpenWrite);
  try
    XMLDoc.Write(XML, Length(XML));
  finally
    XMLDoc.Free;
  end;
end;
```

## Applying updates from an XML document to a provider

*TXMLTransformClient* also lets you insert all of the data from an XML document into the provider's dataset or to delete all of the records in an XML document from the provider's dataset. To perform these updates, call the *ApplyUpdates* method, passing in

- A string whose value is the contents of the XML document with the data to insert or delete.
- The name of a transformation file that can convert that XML data into an insert or delete delta packet. (When you define the transformation file using the XML mapper utility, you specify whether the transformation is for an insert or delete delta packet.)
- The number of update errors that can be tolerated before the update operation is aborted. If fewer than the specified number of records can't be inserted or deleted, *ApplyUpdates* returns the number of actual failures. If more than the specified number of records can't be inserted or deleted, the entire update operation is rolled back, and no update is performed.

The following call transforms the XML document Customers.xml into a delta packet and applies all updates regardless of the number of errors:

```
StringList1.LoadFromFile('Customers.xml');
nErrors := ApplyUpdates(StringList1.Text, 'CustXMLToInsert.xtr', -1);
```