Database Application Developer's Guide

Delphi for Windows

Chapter 1 Introduction

Copyright Agreement

Delphi enables you to create robust database applications quickly and easily. Delphi database applications can work directly with desktop databases like Paradox, dBASE, the Local InterBase Server, and ODBC data sources. The Delphi Client/Server edition also works with remote database servers such as Oracle, Sybase, Microsoft SQL Server, Informix, InterBase, and ODBC data sources. Delphi client applications can be scaled easily between mission critical network-based client/server databases, and local databases on a single machine.

This chapter introduces Delphi's database tools, including the Data Access and Data Controls component pages, the Fields Editor, the Database Desktop, and the Database Forms Expert.

What you should know first

Building a database application is similar to building any other Delphi application. This book assumes you understand the basic application development techniques covered in the Delphi *User's Guide*, including:

- Creating and managing *projects*
- Creating *forms* and managing *units*
- Working with *components*, *properties*, and *events*
- Writing simple Object Pascal source code

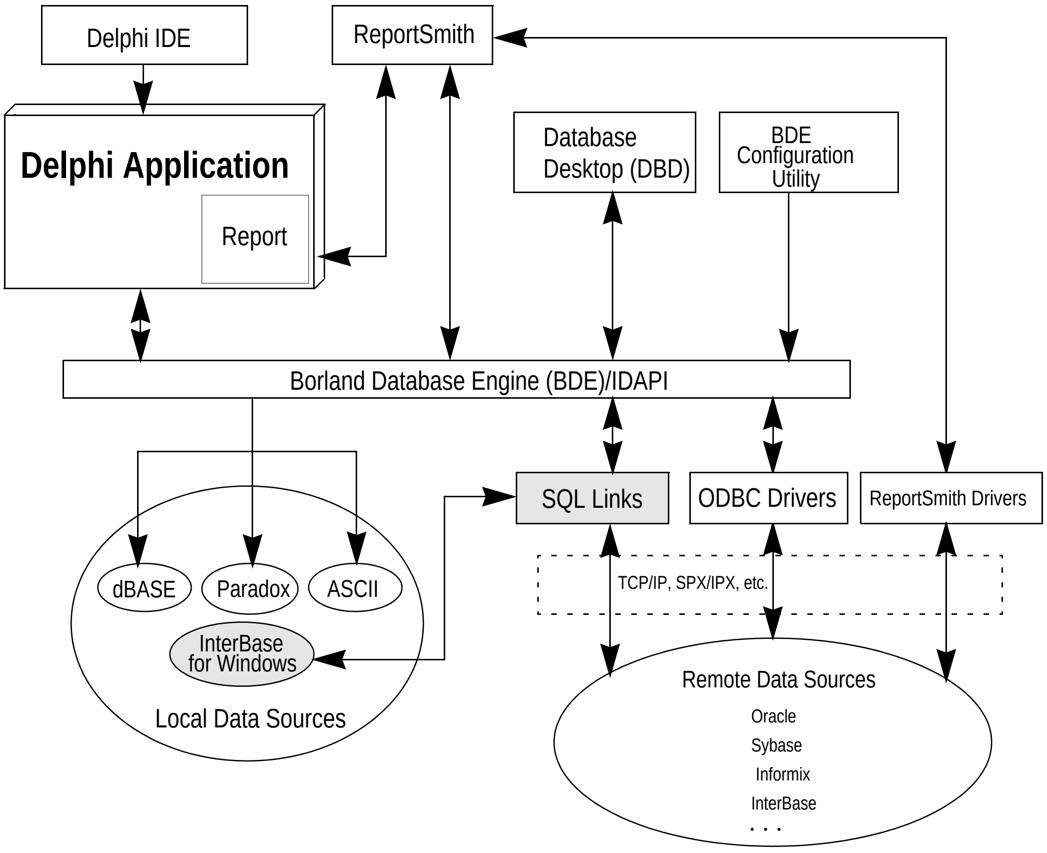
You also need to have a working knowledge of the Database Management System (DBMS) your Delphi database applications access, whether it is a desktop database such as dBASE or Paradox, or an SQL server. For information specific to building client/server applications with Delphi, see Chapter 6, "Building a client/server application."

This book assumes you have a basic understanding of relational databases, database design, and data management. There are many third-party books covering these topics if you need to learn more about them.

Overview of Delphi's database features and capabilities

A Delphi database application is built using Delphi database development tools, Delphi data-access components, and data-aware GUI components. A database application uses Delphi components to communicate with the Borland Database Engine (BDE), which in turn communicates with databases. The following figure illustrates the relationship of Delphi tools and Delphi database applications to the BDE and data sources:

Figure 1.1 Delphi database architecture



The following table summarizes Delphi's database features.

Table 1.1 Database features summary

Tool	Purpose
Data Access components	Access databases, tables, stored procedures, and custom component editors.
Data Control components	Provide user interface to database tables.
Database Desktop (DBD)	Create, index, and query Paradox and dBASE tables, and SQL databases. Access and edit data from all sources.

Table 1.1 Database features summary (continued)

Tool	Purpose
ReportSmith	Create, view, and print reports.
Borland Database Engine (BDE)	Access data from file-based Paradox and dBASE tables, and from local InterBase server databases.
BDE Configuration Utility	Create and manage database connection Aliases used by the BDE.
Local InterBase Server	Provides a single-user, multi-instance desktop SQL server for building and testing Delphi applications, before scaling them up to a production database, such as Oracle, Sybase, Informix, or InterBase on a remote server.
InterBase SQL Link	Native driver that connect Delphi applications to the Local InterBase Server.

These features enable you to build database applications with live connections to Paradox and dBASE tables, and the Local InterBase Server through the BDE. In many cases, you can create simple data access applications with these components and their properties without writing a line of code.

The BDE is built into Delphi components so you can create database applications without needing to know anything about the BDE. The Delphi installation program installs drivers and sets up configuration for Paradox, dBASE, and the Local InterBase Server, so you can begin working with tables native to these systems immediately. The BDE Configuration Utility enables you to tailor database connections and manage database aliases.

Advanced BDE features are available to programmers who need more functionality. These features include local SQL, which is a subset of the industry-standard SQL that enables you to issue SQL statements against Paradox and dBASE tables; low-level API function calls for direct engine access; and ODBC support for communication with other ODBC-compliant databases, such as Access and Btrieve.

Delphi includes Borland ReportSmith, so you can embed database report creation, viewing, and printing capabilities in Delphi database applications. Delphi also includes the Database Desktop (DBD), a tool that enables you to create, index, and query desktop and SQL databases, and to copy data from one source to another. For more information about ReportSmith, see *Creating Reports*. For more information about the DBD, see Appendix A, "Using Database Desktop."

The Local InterBase Server is a single-user, multi-instance, 16-bit, ANSI SQL-compliant, Windows-based version of Borland's 32-bit InterBase SQL server that is available for Novell NetWare, Windows NT, and Unix. For more information, see the *Local InterBase Server User's Guide*.

The following table lists the additional database features available in the Client/server edition of Delphi. These features extend Delphi's database capabilities to access remote

SQL database servers such as Sybase, Microsoft SQL Server, Oracle, Informix, and InterBase.

Table 1.2 Additional Delphi Client/Server database features

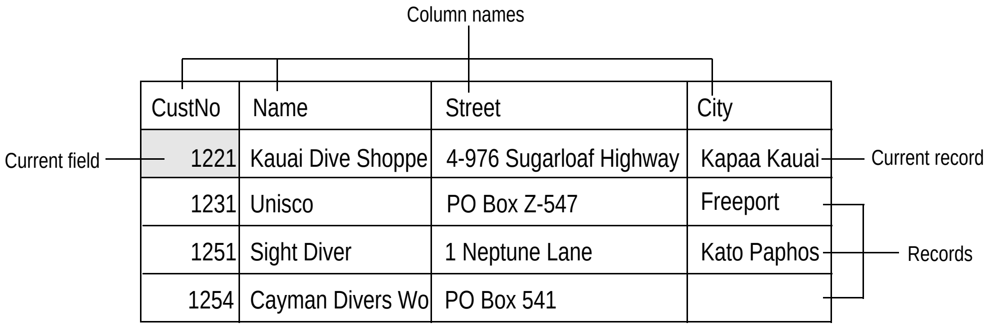
Tool	Purpose
SQL Drivers	Both SQL Links and ReportSmith provide native drivers that connect Delphi database applications to remote SQL database servers, such as Oracle, Sybase, Microsoft SQL Server, Informix, and InterBase.
Visual Query Builder	Creates SQL statements by visually manipulating tables and columns.

SQL Links provide Delphi applications with SQL access to data residing on remote servers, including Sybase, Microsoft SQL Server, Oracle, and Informix. When an SQL Link driver is installed, SQL statements are passed directly to the server for parsing and execution. For more information about using passthrough SQL, see Chapter 5, "Using SQL in applications."

What is a database?

Delphi programmers should understand some basic concepts about databases, data, and data access, before building database applications. A database consists of one or more tables, where each table contains a series of columns into which records (also called "rows") are stored. Each record is identical in structure. For example, a database of addresses consists of a table with name, street address, city, state, and zipcode columns. The intersection of a single column and row is referred to as a field. Fields contain values. The following figure illustrates these concepts:

Figure 1.2 Structure of a table



The current *field* is one field in a single record. The current *record* is a single record in a multi-record set that is the focus of attention. For example, some Delphi database applications display multiple columns and records in a grid format for editing. As far as Delphi controls are concerned, only one field in a single record is "current," meaning that editing tasks affect only the data in that field.

Different databases vary widely in structure. A database in Paradox consists of one or more files, each of which contains a single table or index, but an SQL relational database on a remote server generally consists of a single file that contains all tables, indices, and other database structures. Delphi's Data Access and Data Control components encapsulate the underlying structure of the databases your application uses, so that your application can present the same interface to an end user whether it accesses a local Paradox file or a database on a remote SQL server.

The information in most databases is constantly changing. When you access a networked database from a Delphi application, many users may be accessing and updating the database at the same time. When any database application accesses a database, whether to process a query or generate a report, the application receives a snapshot of the database as it was at the time the application accessed the database. An application's view of data may differ from the data currently in the database, so database applications should always be robust enough to react to such data changes. For more information about building client/server applications that access remote data, see Chapter 6, "Building a client/server application."

What is data?

In this book, "data" refers to information stored in a database. Data may be a single item in a field, a record that consists of a series of fields, or a set of records. Delphi applications can retrieve, add, modify, or delete data in a database.

What is data access?

Delphi applications can access data from desktop database tables on a file server or local disk drive and from remote database servers. To access a data source, a Delphi application uses Data Access components to establish a connection through the BDE. The installation program for Delphi installs drivers and sets up configurations for Paradox, dBASE, and the Local InterBase Server so you can begin working with tables native to these systems immediately.

To connect to another data source requires the installation of a driver for that specific database and subsequent configuration of the BDE to recognize the driver. Connecting to remote database servers requires the Client/Server edition of Delphi that includes SQL Links to access to Sybase, Microsoft SQL Server, Oracle, Informix, and InterBase on NT, NetWare, and Unix servers. For more information about installing and configuring the SQL Link drivers, see the *SQL Links User's Guide*.

The BDE uses *aliases* as convenient shorthand names for often-used data sources, whether local or remote. The BDE Configuration Utility enables you to define and modify aliases that Delphi applications can use immediately. For more information about defining aliases, see Appendix B, "Using the BDE configuration utility."

Once drivers are installed and network connections established, Delphi applications can access data from any authorized server. The examples in Chapter 2 demonstrate techniques for accessing data from a database—specifically, sample data tables that are shipped and installed as part of the Delphi package. Although the example project in the next chapter deals with local desktop data, the techniques for accessing remote data are essentially the same, as subsequent chapters demonstrate.

Note An SQL version of the example project is provided in the DEMOS directory.

Data sources

Delphi database applications get their data through the BDE. The different data sources (not to be confused with the *TDataSource* component) that the BDE can use are shown in Table 1.3.

Table 1.3 Delphi data sources

Data source	Description	File extension
Paradox	Tables created with Paradox or Database Desktop. Each table is in a separate file.	.DB
dBASE	Tables created with dBASE or Database Desktop. Each table is in a separate file.	.DBF
ASCII files	Tables created with Database Desktop. Each table is in a separate file.	.TXT
Local InterBase Server	Database created with InterBase Windows ISQL. Multiple tables in a single database file.	.GDB
SQL Database Server: Oracle, Sybase, Microsoft SQL Server, Informix, InterBase	Database created with server-specific tools, or the DBD, accessed across network with SQL Links. Delphi Client/Server Edition only.	Depends on server
ODBC data sources	Databases such as Microsoft Access, Btrieve, FoxPro, etc.	Depends on data source

Understanding Delphi database architecture

Delphi uses object-oriented components to create database applications, just as it does with non-database applications. Like standard components, database components have attributes, or *properties*, that are set by the programmer at design time. These properties can also be set programmatically at run time.

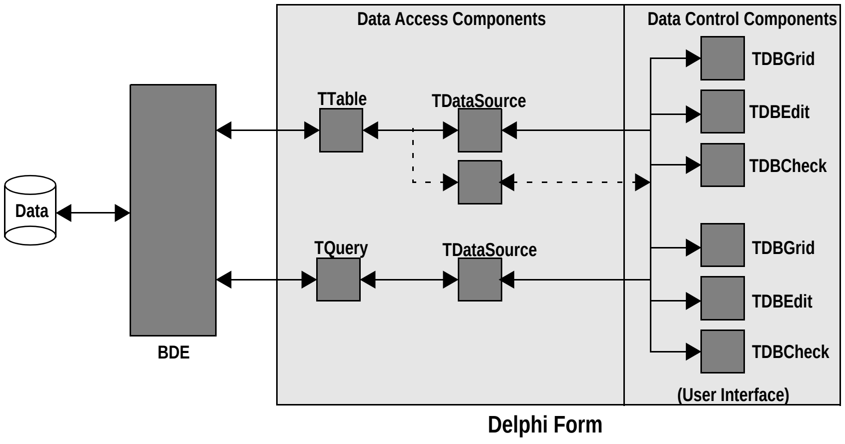
Database components have default behavior that enables them to perform useful functions with little or no programming. The Delphi Component palette provides two database component pages:

- The *Data Access page* contains Delphi objects that simplify database access by encapsulating database source information, such as the database to connect to, the tables in that database to access, and specific field references within those tables. Examples of the most frequently used data access objects include *TTable*, *TQuery*, *TDataSource*, and *TReport*.
- The *Data Controls page* contains data-aware user interface components for displaying database information in forms. Data Control components are like standard user interface components, except that their contents can be derived from or passed to database tables. Examples of the most frequently used data control components include *TDBEdit*, *TDBNavigator*, and *TDBGrid*.

Datasets, such as *TTable*, *TQuery*, and *TStoredPro*c components, are not visible at run time, but provide applications their connection to data through the BDE. Data Control components are attached to dataset components by a *TDataSource* component, to provide a visual interface to data.

The following figure illustrates how Data Access and Data Control components relate to data, to one another, and to the user interface in a Delphi database application:

Figure 1.3 Database components architecture

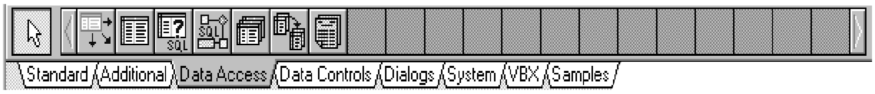


As this figure illustrates, a form usually contains at least three database components: a dataset component (*TTable* and *TQuery* in the figure) that communicates with the BDE; a *TDataSource* component that acts as a conduit between a dataset component and the user interface; and one or more data control components, such as *TDBEdit* or *TDBGrid*, that enable a user to browse, edit, or enter data.

Overview of the Data Access page

The Data Access page of the Delphi Component palette provides a set of database encapsulation objects that simplify database access.

Figure 1.4 Data Access page of the Component palette



When building a database application, you place data access components on a form, then assign them properties that specify the database, table, and records to access. They provide the connection between a data source and Data Control components.

At run time, after an application is built and compiled, data access objects are not visible, but are "under the hood," where they manage data access.

The following table lists the data access objects on the Data Access page, and briefly describes how they are used:

Table 1.4 Data Access components

Component	Purpose
TDataSource	Acts as a conduit between a TTable, TQuery, TStoredProc component and data-aware components, such as TDBGrid.
TTable	Retrieves data from a database table via the BDE and supplies it to one or more data-aware components through a *TDataSource* component. Sends data received from a component to a database via the BDE.
TQuery	Uses SQL statements to retrieve data from a database table via the BDE and supplies it to one or more data-aware components through a *TDataSource* component, or uses SQL statements to send data from a component to a database via the BDE.
TStoredProc	Enables an application to access server stored procedures. Sends data received from a component to a database via the BDE.
TDatabase	Sets up a persistent connection to a database, especially a remote database requiring a user login and password.
TBatchMove	Copies a table structure or its data. Can be used to move entire tables from one database format to another.
TReport	Enables printing and viewing of database reports through ReportSmith.

Four data access components deserve special mention. Most forms provide a link to a database with a *TTable* or *TQuery* component (or through a user-defined component based on the normally hidden abstract class, *TDataSet*, of which *TTable* and *TQuery* are descendents). Other forms provide a link to a database with *TStoredProc*, also a descendent of *TDataSet*. In turn, all forms must provide a *TDataSource* component to link a *TTable*, *TQuery*, or *TStoredProc* component to data control components that provide the visible user interface to the data.

TTable, *TQuery*, (and *TStoredProc*, when it returns a result set) contain a collection of *TField* components. Each *TField* corresponds to a column or field in the table or query. *TFields* are created

- Automatically, when *TTable*, *TQuery*, or *TStoredProc* are activated.
- At design time, using the Fields editor.

For more information about *TFields* and the Fields editor, see Chapter 3, "Using data access components and tools." For more information about *TStoredProc*, see Chapter 6, "Building a client/server application."

Understanding TTable

The *TTable* component is the easiest way for a programmer to specify a database table for access. To put a *TTable* component on a form:

1. Select the Data Access page from the Component palette.
2. Click the Table icon.
3. Click on the form to drop the *TTable* component.
4. Enter the directory where the database resides in the *DatabaseName* property of the Object Inspector window. For SQL databases, enter an alias name.

Note An alias can also be used for local Paradox and dBASE tables. You can choose an alias from a drop-down list in the Object Inspector.

5. Enter the name of the table to use in the *TableName* property of the Object Inspector window, or you can also choose a table from the drop-down list instead of entering the name.

By default, a *TTable* component accesses every column in a table when you activate it. When a visual component, such as *TDBEdit*, is associated with a *TTable* object, it can display any field in the table. Multi-column visual components, such as *TDBGrid*, access and display columns in the table using the table's *TField* list.

If you double-click a *TTable* component on a form, you invoke the Fields Editor. The Fields Editor enables you to control the way Data Control components display data. It can

- Create a static model of a table's columns, column order, and column type that does not change even if changes are made to the underlying physical table in the database.
- Provide convenient, readable, and efficient component names for programmatic access.
- Specify the order in which fields are displayed and which fields to include.
- Specify all display characteristics of fields.
- Add custom validation code.
- Create new fields for display, including calculated fields.

For complete information about the Fields Editor, see Chapter 3, "Using data access components and tools."

Understanding TQuery

The *TQuery* component provides a tool for data access using SQL statements, such as a SELECT statement, to specify a set of records and a subset of columns from a table. *TQuery* is useful for building local SQL queries against Paradox and dBASE data, and for building client/server applications that run against SQL servers.

To put a *TQuery* component on a form:

1. Select the Data Access page from the Component palette.
2. Choose the Query icon.
3. Click on the form to drop the *TQuery* component.
4. Enter the directory where the database resides (or select an alias for SQL databases) in the *DatabaseName* property of the Object Inspector window.
5. Enter the SQL statement to use for data access in the SQL property of the Object Inspector window by clicking the list button to open the String Editor.

The Object Inspector window for *TQuery* does not contain a separate property for specifying a table name. Instead, a table name must always specified as part of the SQL statement in the SQL property.

With Delphi Client/Server, you can right-click a *TQuery* component on a form, then select the Visual Query Builder from the pop-up menu. The Visual Query Builder enables you to connect to a database and build an SQL statement interactively. For complete information about the Visual Query Builder, see the online Help.

If you double-click a *TQuery* component, you invoke the Fields Editor. The Fields Editor enables you to control the way Data Control components display data. For complete information on *TQuery* and the Fields Editor, see Chapter 3, "Using data access components and tools."

Understanding TDataSource

Every dataset that supplies a data control component must have at least one *TDataSource* component. *TDataSource* acts as a bridge between one *TTable*, *TQuery*, or *TStoredProc* component and one or more data control components that provide a visible user interface to data.

TTable and *TQuery* can establish connections to a database through the BDE, but they cannot display database information on a form. Data Control components provide the visible user interface to data, but are unaware of the structure of the table from which they receive (and to which they send) data. A *TDataSource* component bridges the gap.

To put a *TDataSource* component on a form:

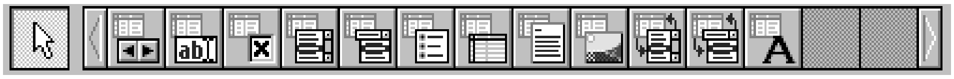
1. Select the Data Access page from the Component palette.
2. Choose the DataSource icon.
3. Click on the form to create the *TDataSource* component.
4. Enter the name of the *TTable* or *TQuery* component to use as a database connection source in the DataSet property of the Object Inspector. If the form contains any *TTable* or *TQuery* components, you can choose a component from the drop-down list instead.

Note TDataSource is also used to link tables or queries in a master/detail form. For more information about master/detail forms, see Chapter 2, "Building a sample database application: MASTAPP."

Overview of the Data Controls page

The Data Controls page provides a set of data-aware user-interface components that you can use to create forms-based database applications.

Figure 1.5 The Data Controls page of the Component palette



Many data controls are data-aware versions of component classes available on the Standard page of the Component palette. In addition to standard component functionality, data controls can display data from a field in a database table, or send new or modified data from a form to a database table.

The following table lists the data controls on the Data Control page.

Table 1.5 Data Controls components

Component	Purpose
TDBNavigator	Data-aware navigation buttons that move a table's current record pointer forward or backward; start Insert or Edit mode; post new or modified records; cancel Edit mode; and refresh display to retrieve updated data.
TDBText	Data-aware label that can display a field from a currently active record.
TDBEdit	Data-aware edit box that can display or edit a field from a currently active record.
TDBCheckBox	Data-aware check box that can display or edit a Boolean data field from a currently active record.
TDBListBox	Data-aware list box that can display values from a column in a table.
TDBComboBox	Data-aware combo box that can display or edit values from a column in a table.
TDBRadioGroup	Data-aware radio group populated with radio buttons that can display or set column values.
TDBGrid	Data-aware custom grid that enables viewing and editing data in a tabular form similar to a spreadsheet; makes extensive use of *TField* properties (set in the Fields Editor) to determine a column's visibility, display format, ordering, etc.
TDBMemo	Data-aware memo box that can display or edit text BLOB data from a currently active record.
TDBImage	Data-aware image box that can display, cut, or paste bitmapped BLOB images to and from a currently active record.
TDBLookupList	Data-aware list box that displays values mapped through another table at run time.
TDBLookupCombo	Data-aware combo box that displays values mapped through another table at run time.

Data control components make up a consistent visual user interface for Delphi database applications, whether the application accesses a local database file, or a remote database server. To see how data control components are used in an application, see the subsequent chapters of this book. For a complete description of each data control component and its properties, see the online VCL Reference.

Overview of the Database Forms Expert

The Database Forms Expert automates many of the tasks necessary for creating data-entry or tabular forms from an existing database table. It can generate simple or master/detail forms using *TTable* or *TQuery* components. The Database Forms Expert automates such form building tasks as:

- Placing database components on a form.
- Connecting *TDataSet* components (e.g., *TTable* and *TQuery*) to a database.
- Connecting *TDataSource* components to interactive data control components and *TTable* or *TQuery* data access objects.
- Writing SQL statements for *TQuery* objects.
- Defining a tab order for components.

Inexperienced database applications programmers can use the Database Forms Expert to learn how to build database forms. Experienced database applications programmers can use it to speed application development. To learn how to use the Database Forms Expert when building an application, see Chapter 2, “Building a sample database application: MASTAPP.”

Overview of the Database Desktop

The Database Desktop (DBD) is a database maintenance and data definition tool. It enables programmers to query, create, restructure, index, modify, and copy database tables, including Paradox and dBASE files, and SQL tables. You do not have to own Paradox or dBASE to use the DBD with desktop files in these formats.

The DBD can copy data and data dictionary information from one format to another. For example, you can copy a Paradox table to an existing database on a remote SQL server. For a complete description of the DBD, see Appendix A, “Using Database Desktop.”

Developing applications for desktop and remote servers

Delphi Client/Server enables programmers to develop and deploy database client applications for both desktop and remote servers. One of Delphi’s strengths is the ease with which an application developed for the desktop can be adapted to access data on a remote SQL server. The user interface need not change even if the source of the data changes. To an end user, a Delphi database application looks the same whether it accesses a local database file or a remote SQL database.

For simple applications that use *TQuery* components to access desktop data, the transition to a remote server may be as simple as changing the data source. For other applications, more significant changes may be in order. Some of these changes are the result of differing conventions and concurrency issues between desktop and SQL databases.

For example, desktop databases like Paradox and dBASE are record-oriented. They always display records in ascending or descending alphabetic or numeric order. They lock and access a single record at a time. Each time a user changes a record, the changes are immediately written to the database. Desktop database users can see a range of records, and can efficiently navigate forward and backward through that range.

In contrast, data in SQL databases is set-oriented, and designed for simultaneous multiuser access. Record ordering must be specified as part of an SQL query. To accommodate multiuser access to data, SQL relies on transactions to govern access. For more information about working with transactions, see Chapter 6, “Building a client/server application.”

Database application development methodology

Developing database applications with Delphi is similar to developing other types of software, but there are important distinctions and challenges that must be addressed. The methodology presented in this section should be used as a guideline that you can adapt to meet your specific business needs.

Development scenarios

Since an application's design usually depends on the structure of the database it will access, the database must be defined before the application can be developed.

Note Database development (also called *data definition*) is a part of the overall development process, but is beyond the scope of this manual. For more information, refer to the numerous books about relational database design.

There are four possible scenarios for Delphi database application development:

- The database does not yet exist or must be re-defined.
  - Use the Database Desktop utility to define Paradox and dBASE tables. For more information, see Appendix A, "Using Database Desktop."
  - For SQL servers, use the tools provided with the server or the Database Desktop. For example, for the Local InterBase Server or an InterBase Workgroup Server, use Windows ISQL. For more information, see the *Local InterBase Server User's Guide* and the InterBase *Data Definition Guide*.
- The database exists on a desktop or LAN data source (Paradox or dBASE) and the database will access it there. If the BDE and the data source are on the same machine as the application, then the application is a standalone (not client/server) application.
- The database exists on a desktop data source, and is being upsized to an SQL server. This scenario is discussed in Appendix C, "Using local SQL."
- The database exists on an SQL server and the application will access it there. This is a standard client/server application. For information specific to developing a client/ server application, Chapter 6, "Building a client/server application."

Database application development cycle

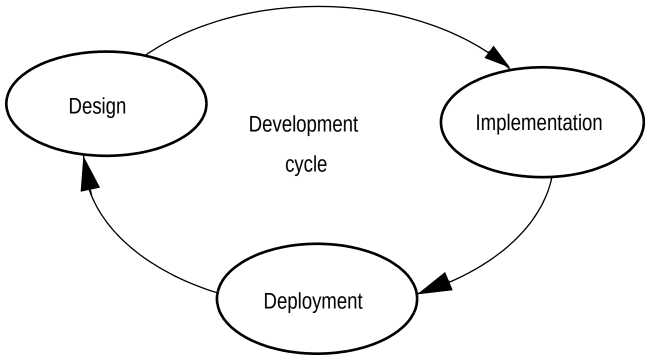
The goal of database application development is to build a product which meets end users' long-term needs. While this goal may seem obvious, it is important not to lose sight of it throughout the complexities and often conflicting demands of the development process. To create a successful application it is critical to define the end users' needs in detail early in the development process.

The three primary stages of database application development are

- Design and prototyping
- Implementation
- Deployment and maintenance

There are database and application tasks in each of these phases. Depending on the size and scope of the development project, the database and application tasks may be performed by different individuals or by the same individual. Often, one team or individual will be responsible for the database tasks of the project, and another team or individual will be responsible for the application tasks.

Figure 1.6 Development cycle



For client/server applications, the database and application tasks become more distinct, since they run on different platforms, often with different operating systems (for example, a Unix server and Windows 3.1 client).

When development responsibilities are thus divided it is important to clearly delineate in the design phase which functions will be performed by the database server and which will be performed by the client application. Usually, the functional lines are clear cut. But database processes such as stored procedures can sometimes perform functions that can also be performed by the client application. Depending on the expected deployment configuration, application requirements, and other considerations, the design can allocate such functions to either client or server.

It is also important to realize that database application development is by its nature an iterative process. Users may not fully understand their own needs, or may define additional needs as development proceeds. User interface elements are always refined as they are used. Also, changing business needs will change requirements over time. Generally, a number of iterations through the development cycle will be required before an application can meet a significant portion of its requirements.

Design phase

The design phase begins with requirements definition. In consultation with knowledgeable end users, define the functional specifications for the database and applications. Determine which aspects of the functional requirements will be implemented in the database design, and which aspects will be implemented in the applications.

For client/server applications, often certain functions can be performed either by the server or by the application; for example, a complex mathematical transform function could be performed either by the client application or by a stored procedure on the server. The hardware deployment configuration will generally determine whether such functions are best performed on the server or client. For example, if the client platforms are expected to be low-end desktop PCs, and the server platform is expected to be a high-end workstation, then it will probably be best to run computation-intensive functions on the server. If the hardware configuration changes, then it is possible to move the function between client and server in a later iteration.

Implementation phase

In the implementation phase, you use Delphi to build and test the application conceived in the design phase. During the implementation phase, you should use a duplicate data source, that is, a data source that has the same essential structure as the production database, but with a small subset of representative data. It is not recommended to develop an application against a production database, since the untested application may corrupt the data or otherwise interfere with normal database activities.

If your application will ultimately be deployed to use a desktop data source, make copies of the required tables with the Database Desktop, and populate them with representative "dummy" data.

If the application will ultimately be deployed to use a remote data source (an SQL server), then you can take two approaches during the implementation phase:

- Develop and test the application against a non-production database on the Local InterBase Server.
- Develop and test the application against a non-production database on the server.

The first approach has the advantage that is isolated on the development platform(s), and so will not interfere with other server activities. It will not consume server resources or increase network traffic. Its primary disadvantage is that only standard SQL server features can be used and tested during this phase, if you are using a server other than InterBase for the deployed application.

The second approach enables you to surface all server-specific features, but will consume network and server resources during testing. This approach can be dangerous, since it is conceivable that a programmer error could cause a server to crash during testing.

Deployment phase

In the deployment phase, the client/server application is put to the acid test: it is handed over to end users. To ensure that the application's basic functionality is error-free, deploy a prototype application before attempting to deploy a production application.

Since the ultimate judges of an application's efficacy are its users, developers must be prepared to incorporate changes to applications arising from their suggestions, changing business needs, and for general enhancement (for example, for usability). Sometimes application changes may require changes to the database, and conversely, changes to the database may require application changes. For this reason, application developers and database developers should work together closely during this phase. As features and enhancements are incorporated into the application, the application moves iteratively closer to completion.

Deploying a client/server application requires addressing a number of special issues, including connectivity and multiuser access. These issues are discussed in Chapter 6, "Building a client/server application."

Deploying an application

Deploying an application means giving it to the end users, and providing the necessary software they need to use the application in a production environment. Non-database applications require only an .EXE file to run—Delphi applications do not require a run time interpreter or DLL.

Typically, when deploying a database application, you will create a package that includes all the files that end users need to run the application and access data sources. These files include

- The application .EXE file and .DLL files (if any)
- Required ancillary files (for example, a README file or .HLP files for online help)
- BDE support for database access (desktop or server)
- ReportSmith Runtime for running and printing reports
- If the application uses VBX controls, include each VBX along with BIVBX11.DLL

If you are distributing the files on disks, you will generally want to compress them with a standard file compression utility, and provide the utility on the disk. You may also want to build a simple installation application to install the files for your users. For complex applications, you may want to use one of the many commercially-available installation programs.

Important Before distributing any files, ensure that you have the proper redistribution rights. As described in the Delphi license agreement, Delphi provides distribution rights for the BDE (including Paradox and dBASE support). Delphi Client/Server includes distribution rights for Borland SQL Links for Windows. Licenses for distribution of the Local InterBase Server are available from Borland.

For information on deploying support for remote server access, see Chapter 6, "Building a client/server application." For client/server applications, you also must ensure that the necessary communications software (for example, TCP/IP interface) is installed on the client platforms. This software is provided with databases servers. For more information, see your server documentation.

Deploying BDE support

When you deploy a database application, you must ensure that the client platform has the correct version of the BDE installed. Delphi includes Redistributable BDE, with its own installation utility, that can you can redistribute with your applications. When you deploy an application, simply include a copy of the Redistributable BDE disk.

The Delphi license agreement requires you to make *all* the files in Redistributable BDE available to your application users. This requirement enables users to install the new version of the BDE for Delphi without interfering with existing Paradox and dBASE applications. You can advise your users to save disk space and install only the drivers required to run your application, but you must still distribute all the files in the Redistributable BDE.

For example, if your application needs access only to Paradox files, you can advise your users not to deploy the dBASE driver. The minimum BDE configuration for accessing a Paradox database requires about 500 Kbytes.

Note For more information on deployment, refer to the file DEPLOY.TXT installed to the DELPHI\DOC directory by default.

Table 1.6 Redistributable Borland Database Engine files

File name	Description
IDAPI01.DLL	BDE API DLL
IDBAT01.DLL	BDE Batch Utilities DLL
IDQRY01.DLL	BDE Query DLL
IDASCI01.DLL	BDE ASCII Driver DLL
IDPDX01.DLL	BDE Paradox Driver DLL
IDDBAS01.DLL	BDE dBASE Driver DLL
IDR10009.DLL	BDE Resources DLL
ILD01.DLL	Language Driver DLL
IDODBC01.DLL	BDE ODBC Socket DLL
ODBC.NEW	Microsoft ODBC Driver Manager DLL, version 2.0
ODBCINST.NEW	Microsoft ODBC Driver installation DLL, version 2.0
TUTILITY.DLL	BDE Tutility DLL
BDECFG.EXE	BDE Configuration Utility
BDECFG.HLP	BDE Configuration Utility Help
IDAPI.CFG	BDE (IDAPI) Configuration File

Language drivers

The BDE provides the ability to localize applications with language drivers. The language driver DLL loads the drivers specified by Paradox or dBASE tables or in IDAPI.CFG for server databases. The language drivers are files with extension .LD installed to the LANGDRV sub-directory of the BDE directory.

Important For language drivers to load correctly, the WIN.INI file must have the following entry, assuming the default installation directory:

```
[Borland Language Drivers]
LDPath = C:\DELPHI\IDAPI\LANGDRV
```

ODBC Socket

The BDE comes with an ODBC Socket. It has been certified with Microsoft's 2.0 ODBC Driver Manager. If you have a different version of the ODBC Driver Manager:

- Back up your existing ODBC.DLL and ODBCINST.DLL
- Copy the version 2.0 files, ODBC.NEW and ODBCINST.NEW, from your BDE directory to your WINDOWS\SYSTEM directory.
- Rename these files to ODBC.DLL and ODBCINST.DLL.

Note The ODBC 2.0 Driver Manager does work with ODBC 1.x ODBC drivers.

Chapter

2

Building a sample database application: MASTAPP

This chapter is a tutorial and introduction to building Delphi database applications. Examples show how to perform database tasks using Delphi interactively and by programming in Object Pascal. In each example you build a single form, self-contained and independent of the others. You can save yourself some work by doing the examples in sequence. Several examples use the same basic form as a starting point.

Note This material assumes you know how to use Delphi; it tells you *what* to do to perform certain tasks. For more details (that is, to find out *why*), follow the cross-references in the "For more information" section that follows each example. In particular, see Chapter 3, "Using data access components and tools."

The tutorial consists of the following sections:

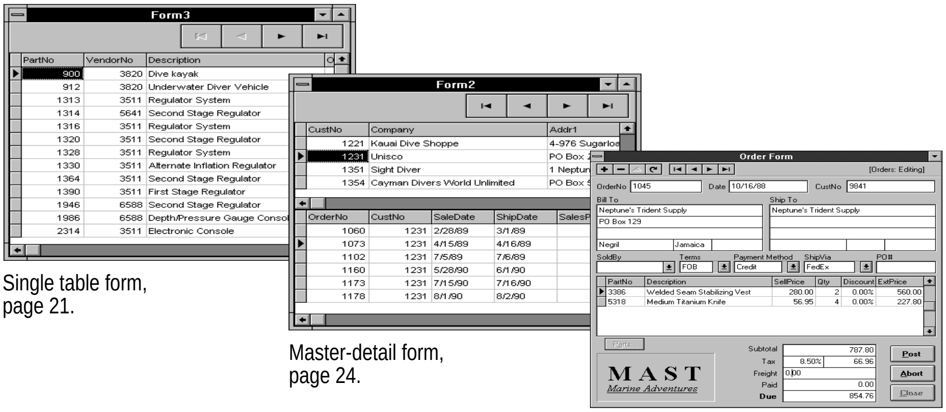
- "Building forms" describes how to use the Database Form Expert to create database forms, including single-table and master-detail forms. It also describes how to enhance forms by adding components and code by hand.
- "Working with fields" describes how to read and write field values, how to search for values and do table lookups, and how to format data displayed to the user. It also describes how to work with calculated fields.
- "Using queries and ranges" describes how to use SQL queries and set ranges to select a subset of the data in one or more tables.
- "Printing reports and forms" describes how to print ReportSmith reports and Delphi forms.

Building forms

The material in this section focuses on database issues. To learn about general application building with Delphi, see the *User's Guide*. The forms described here are the basis for a database application called MASTAPP, designed to meet the record-keeping needs of the fictitious Marine Adventures & Sunken Treasures company (MAST). MAST sells diving equipment and arranges diving expeditions. MASTAPP tracks information about customers, orders, inventory, and vendors.

The tutorial starts with a simple "codeless" form for viewing and editing table data, and works up to a full-featured invoice form containing several tables, data-aware components, and other advanced Delphi features. All the forms, tables, and related files are installed by default in C:\DELPHI\DEMOS\DB\MASTAPP. During a default installation of Delphi, an alias, DBDEMOS, that points to the MASTAPP directory is created for you, or you can create your own alias using the BDE configuration utility (see Appendix B). The following figure shows the forms and tells where they are described in this chapter:

Figure 2.1 Database forms described in the tutorial



PartNo	VendorNo	Description
900	3820	Dive kayak
912	3820	Underwater Diver Vehicle
1313	3511	Regulator System
1314	5641	Second Stage Regulator
1316	3511	Regulator System
1320	3511	Second Stage Regulator
1328	3511	Regulator System
1330	3511	Alternate Inflation Regulator
1364	3511	Second Stage Regulator
1390	3511	First Stage Regulator
1946	6588	Second Stage Regulator
1986	6588	Depth/Pressure Gauge Consol
2314	3511	Electronic Console

Single table form, page 21.

CustNo	Company	Addr1
1221	Kauai Dive Shoppe	4-976 Sugarloa
1231	Unisco	PO Box Z
1351	Sight Diver	1 Neptun
1354	Cayman Divers World Unlimited	PO Box 5

OrderNo	CustNo	SaleDate	ShipDate	SalesP
1060	1231	2/28/89	3/1/89	
1073	1231	4/15/89	4/16/89	
1102	1231	7/5/89	7/6/89	
1160	1231	5/28/90	6/1/90	
1173	1231	7/15/90	7/16/90	
1178	1231	8/1/90	8/2/90	

Master-detail form, page 24.

[Orders: Editing]

OrderNo 1045 Date 10/16/88 CustNo 9841

Bill To: Neptune's Trident Supply, PO Box 129, Negril, Jamaica

Ship To: Neptune's Trident Supply

SoldBy: Terms: FOB Payment Method: Credit ShipVia: FedEx PO#:

PartNo	Description	SellPrice	Qty	Discount	ExtPrice
3386	Welded Seam Stabilizing Vest	280.00	2	0.00%	560.00
5318	Medium Titanium Knife	56.95	4	0.00%	227.80

Subtotal		787.80
Tax	8.50%	66.96
Freight	0.00	
Paid		0.00
Due		854.76

MAST *Marine Adventures*

Parts... Post Abort Close

One-many-many form, page 27.

Note For general information about building Delphi forms, see the *User's Guide*.

MASTAPP aliases

All *TTable* and *TQuery* components used in example code in this chapter set their *DatabaseName* property to DBDEMOS. In contrast, the complete demo in the MASTAPP directory does the following to facilitate porting:

1. The main form (MAIN.PAS) has a *TDatabase* component with its *AliasName* property set to DBDEMOS and *DatabaseName* property set to MAST.
2. All datasets on all forms have their *Database* properties set to MAST. Now all forms can use a different BDE alias simply by changing the main form's *TDatabase* component's *AliasName* property.

Building a single-table form

The steps in this section show how to use the Database Form Expert to build a single-table form. Of course, anything the expert does, you can do by hand, but the expert saves a lot of time.

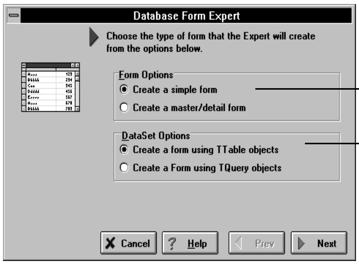
What to do

1 Choose Help | Database Form Expert to open the Form Expert.

2 Specify a table, fields, and field layout as shown in the following figure. The Form Expert creates the form.

3 Press *F9* to run the form. Click the navigator control buttons to move through the records in the table.

Figure 2.2 Building a single-table form using the Database Form Expert

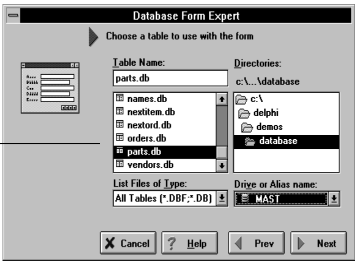

Database Form Expert

Choose the type of form that the Expert will create from the options below.

Form Options
[✓ Create a simple form]
Create a master/detail form

DataSet Options
[✓ Create a form using TTable objects]
Create a Form using TQuery objects

1. Specify a simple form created using TTable objects.


Database Form Expert

Choose a table to use with the form

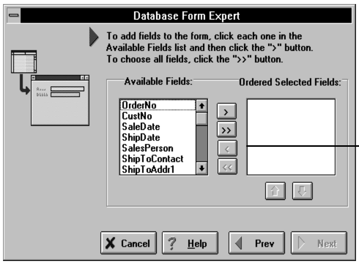
Table Name: parts.db
names.db
nextitem.db
nextord.db
orders.db
[✓ parts.db]
vendors.db

Directories: c:\...\database
c:\
delphi
demos
[✓ database]

List Files of Type: All Tables (*.DBF;*.DB)

Drive or Alias name: MAST

2. Choose a table (PARTS.DB).

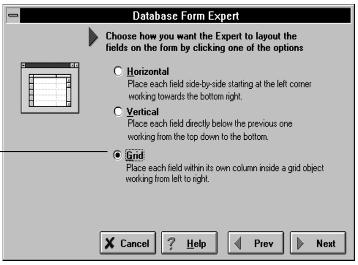

Database Form Expert

To add fields to the form, click each one in the Available Fields list and then click the ">" button. To choose all fields, click the ">>" button.

Available Fields:
OrderNo
CustNo
SaleDate
ShipDate
SalesPerson
ShipToContact
ShipToAddr1

Ordered Selected Fields:

3. Specify which fields to use. Click >> to use all fields.


Database Form Expert

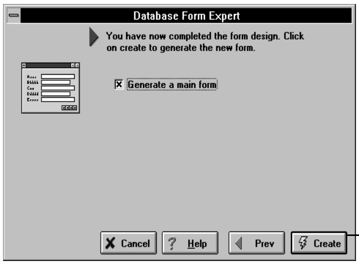
Choose how you want the Expert to layout the fields on the form by clicking one of the options

Horizontal
Place each field side-by-side starting at the left corner working towards the bottom right.

Vertical
Place each field directly below the previous one working from the top down to the bottom.

[✓ Grid]
Place each field within its own column inside a grid object working from left to right.

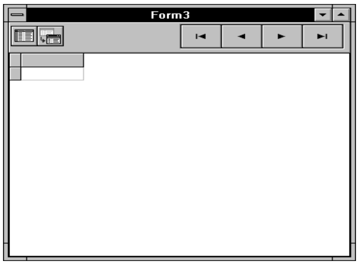
4. Choose a field layout (grid).


Database Form Expert

You have now completed the form design. Click on create to generate the new form.

[✓ Generate a main form]

5. Tell the expert to create the form.

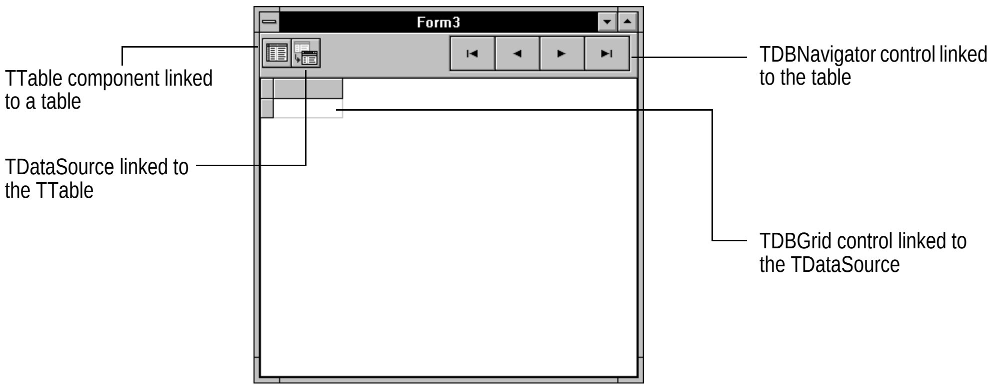


6. This is how the completed form looks by default.

How it works

The Database Form Expert builds a single-table form to match your specifications, and adds a tool bar of navigation controls. The form shown in the following figure is the basis for the full-featured BRPARTS.DFM form in the MASTAPP application. The important relationships in a data-aware form like this one are the links between the underlying data, the nonvisual components, and the data controls that display data to the user.

Figure 2.3 A single-table form


Form3
TTable component linked to a table
TDataSource linked to the TTable
TDBNavigator control linked to the table
TDBGrid control linked to the TDataSource

The form contains one *TTable* component. The expert links it to the *Parts* table by setting the properties listed in the following table. A *TTable* component establishes a connection to a table in a database.

Table 2.1 Important TTable properties for a single-table form

Property	Value	Remarks
Active	False	When *Active* is False, data-aware controls do not display data at design time. To make controls display data at design time, set *Active* to True.
DataBaseName	MAST	MAST is an alias that points to where the table resides. Use aliases, not hard-coded paths, to make applications portable and easy to upsize.
Name	Form1	Use the Object Inspector to change names.
TableName	PARTS.DB	Tells the component which table to link to.

The form contains one *TDataSource* component. The expert links it to the *TTable* component by setting the properties listed in the following table. A *TDataSource* component acts as a bridge between one dataset component (*TTable* in this case) and one or more data-aware controls that provide a visible interface to data.

Table 2.2 Important TDataSource properties for a single-table form

Property	Value	Remarks
AutoEdit	True (default)	When *AutoEdit* is True, Delphi puts the *TDataSource* into Edit state automatically when the user changes a value in a linked control. To make a *TDataSource* read-only, or to control when to enter Edit state, set *AutoEdit* to False.

Table 2.2 Important TDataSource properties for a single-table form (continued)

Property	Value	Remarks
DataSet	Table1	Specifies which *TTable* (or *TQuery*) is supplying the data.
Name	DataSource1	Use the Object Inspector to change names.

The form contains one *TDBGrid* control. The expert links it to a *TDataSource* component by setting the properties listed in the following table. By default, a *TDBGrid* includes all the fields (columns) in a table. To limit the columns displayed by a *TDBGrid*, double-click on its associated *TTable* component to invoke the Fields Editor (see page 30).

Table 2.3 Important TDBGrid properties for a single-table form

Property	Value	Remarks
DataSource	DataSource1	Links the *DBGrid* control to a *TDataSource* component, which supplies the data.

The form contains one *TDBNavigator* control. This control moves a table's current record pointer forward or backward, starts Insert or Edit state, posts new or modified records, etc. The expert links this control to the *Parts* table by setting the properties listed in the following table.

Table 2.4 Important TDBNavigator properties for a single-table form

Property	Value	Remarks
DataSource	DataSource1	Links the control to a *TDataSource* component.
VisibleButtons	A list of button identifiers and a corresponding Boolean value that specifies whether that button is visible. Example: *nbFirst* True.	For example, by default *nbNext* is True, so the Next Record button is visible; *nbDelete* is False, so the Delete button is invisible.

The expert does more than create a form and components. It also generates a line of code to open the table at run time in case you do not activate the table at design time. For example, the expert creates *TTable* components with the *Active* property set to False. That's why the various *TDBEdit* controls aren't displaying data. You could set *Active* to True, and the controls would display data from the first record. Instead, the Form Expert generates the following code to open the table at run time.

```
procedure TEditPartsForm.FormCreate(Sender: TObject);
begin
  Table1.Open;
end;
```

This code is hooked to the form's *OnCreate* event, so Delphi executes it before creating the form. As a result, the table is opened before the form is displayed.

Note Code created by the Form Expert is like code you type yourself. If you rename components created by the expert, be sure to update your source file everywhere the renamed component occurs. Name changes to components that Delphi originates (for example, in the **type** declaration) are changed automatically for you by Delphi.

For more information

For more information about

- Building Delphi forms, see the *User's Guide*.
- Naming components, see the *User's Guide*.
- The Delphi database architecture, see Chapter 1, "Introduction."
- *TTable* components, see page 8.
- The Fields Editor, see page 30.
- *TDataSource* components, see page 10.

Building a master-detail form

The steps in this section show how to use the Database Form Expert to build a form containing two tables: a master table and a detail table, linked one-to-many. This form is the basis for the CUSTORD.DFM form in MASTAPP. The master table is CUSTOMER.DB and the detail table is ORDERS.DB. You can access both tables using the MAST alias. The expert links these tables and creates components to display data for one customer at a time, and for each customer, to display many orders.

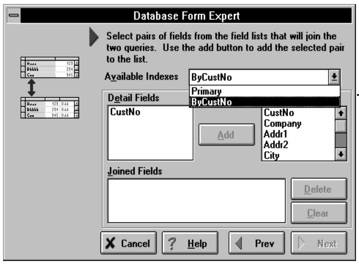
What to do

When you use the Database Form Expert, building a master-detail form is much like building a single-table form (for details, see page 21).

1. Choose Help | Database Form Expert to open the Form Expert.
2. In the first panel, specify a master-detail form that uses *TTable* objects.
3. In subsequent panels, specify the master table (CUSTOMER.DB), fields (use them all), and field layout (grid).
4. Specify the detail table (ORDERS.DB), fields (all), and field layout (grid).
5. Specify fields to link the master and detail tables as shown in the following figure, then tell the expert to create the form.

6 Press *F9* to run the form. Click the navigator control buttons to move through the records in the table.

Figure 2.4 Linking fields in a master-detail form

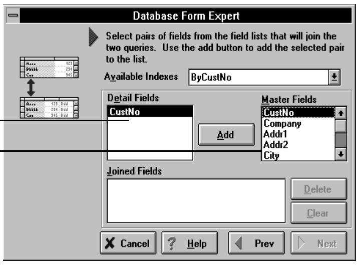


Database Form Expert

Select pairs of fields from the field lists that will join the two queries. Use the add button to add the selected pair to the list.

Available Indexes: ByCustNo

1. Choose an index. All Paradox tables have a primary index by default. ORDERS.DB was also created to have a secondary index named ByCustNo that orders records by customer number. Choose ByCustNo from the combo box.

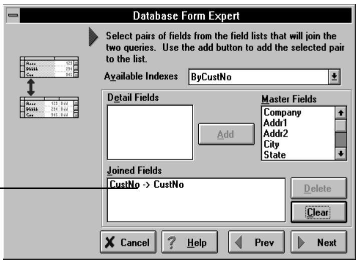


Database Form Expert

Select pairs of fields from the field lists that will join the two queries. Use the add button to add the selected pair to the list.

Available Indexes: ByCustNo

2. When you choose an index, Delphi updates the lists of possible linking fields. In each list, choose CustNo, then click Add.



Database Form Expert

Select pairs of fields from the field lists that will join the two queries. Use the add button to add the selected pair to the list.

Available Indexes: ByCustNo

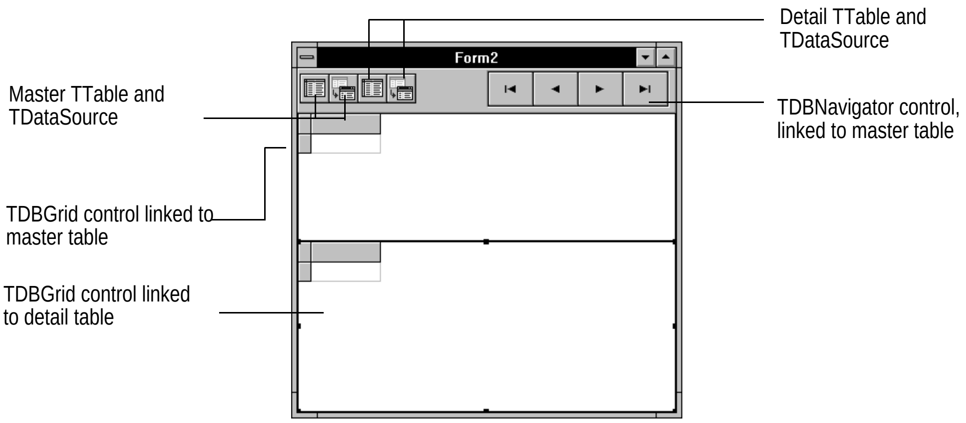
Joined Fields: CustNo -> CustNo

3. When you click Add, Delphi shows how the fields are linked. Click Next to continue.

How it works

The expert builds a master-detail form much as it builds a single-table form (for details, see page 22). It creates *TTable* components and *TDataSource* components for the master table and the detail table and links them to the underlying data by setting properties. The expert creates controls to display the data from each table, and sets properties to link them to the corresponding *TDataSource* component. The expert also creates a *TDBNavigator* control linked to the master table.

Figure 2.5 A master-detail form


Detail TTable and TDataSource
Form2
Master TTable and TDataSource
TDBNavigator control, linked to master table
TDBGrid control linked to master table
TDBGrid control linked to detail table

What distinguishes a master-detail form is the link between the tables. For each record in the master table, the form displays all corresponding records in the detail table. Such a link is called a *one-to-many relationship* (see page 76 for more information). Delphi creates this relationship by setting properties of the *TTable* component linked to the detail table.

Table 2.5 Important detail table properties

Property	Remarks
IndexFieldNames	Specifies the columns used to order the records in the table. You can also specify an index by name using the *IndexFields* property. For more information, see page 74.
MasterFields	Specifies which fields in the master table to link to. Use a semicolon to separate multiple field names. For more information, see page 76.
MasterSource	Specifies the *TDataSource* component linked to the master table. This data source provides a restricted view of the data in the detail table based on the values of the fields specified in *MasterFields*. For more information, see page 76.

The following table lists important properties for each component and control.

Table 2.6 Important component properties for a master-detail form

Component	Property	Value	Remarks
TTable (master)	*Active*	False	When *Active* is False, data from this table is not displayed.
	DataBaseName	MAST	An alias that specifies where to find the table. Use aliases, not hard-coded paths, to make applications portable and easy to upsize.
	Name	Table1 (by default).	Use the Object Inspector to rename it (for example, Cust).
	TableName	CUSTOMER.DB	Specifies the master table.
TDataSource (master)	*DataSet*	Table1 (by default)	Specifies which *TTable* is supplying the data to this component.
	Name	DataSource1 (by default)	Use the Object Inspector to rename it (for example, CustSource).
TTable (detail)	*Active*	False	Data from this table is not displayed at design time.

Table 2.6 Important component properties for a master-detail form (continued)

Component	Property	Value	Remarks
	DataBaseName	MAST	An alias that specifies where to find the table. Use aliases, not hard-coded paths, to make applications portable and easy to upsize.
	IndexFieldNames	CustNo	Specifies a column to use to order records in the table. *ByCustNo* is a secondary index based on *CustNo*. You can also set the *IndexFields* property to *ByCustNo*.
	MasterFields	CustNo	A list of one or more master table fields to link to. Use a semicolon to separate field names in the list.
	MasterSource	DataSource1 (by default)	Identifies a *TDataSource* linked to the master table.
	Name	Table2 (by default)	Use the Object Inspector to rename it (for example, Orders).
	TableName	ORDERS.DB	Specifies the detail table.
TDataSource (detail)	*DataSet*	Table2 (by default)	Specifies which *TTable* is supplying the data to this component.
	Name	DataSource2 (by default)	Use the Object Inspector to rename it (for example, OrdersSource).
TDBGrid (master)	*DataSource*	DataSource1 (by default)	Links the grid control to a *TDataSource* control.
TDBGrid (detail)	*DataSource*	DataSource2 (by default)	This data source provides a restricted view of the data in the *Orders* (detail) table, based on the values of the fields linked to the *Customer* (detail) table.
TDBNavigator	*DataSource*	DataSource1 (by default)	The expert links the *TDBNavigator* control to the master table.

For more information

For more information about

- Linking tables, see step 6 on page 25.
- One to many relationships, see “Creating a master-detail form,” on page 76.
- Using the Database Form Expert, see page 21.

Building a one-many-many form

This section describes how to build a form that displays data from three tables linked one-many-many. For example, one customer may place many orders, and each order may have many items. Use the Database Form Expert to create a master-detail form linking the *Customer* table to the *Orders* table as described on page 24. Then, to display and link the *Items* table, place components and set properties by hand. You can also use the techniques described here to build a master-detail form from scratch.

What to do

1 Choose Help | Database Form Expert to open the Form Expert.

2 In the first panel, specify a master-detail form that uses *TTable* objects.

3. In subsequent panels, specify the master table (CUSTOMER.DB), fields (use them all), and field layout (horizontal).
4. Specify the detail table (ORDERS.DB), fields (all), and field layout (horizontal).
5. Specify fields to link the master and detail tables: choose CustNo for the *IndexFieldNames* property and link the CustNo fields in each table. For details, see step 6 on page 25.
6. Tell the expert to create the form.
7. Move components around to make room at the bottom of the form. You may have to change the *Align* property of some controls from *alClient* to *alNone*.
8. Place a *TTable* component, a *TDataSource* component, and a *TDBGrid* component as shown in the following figure. (The *TTable* and *TDataSource* are on the Data Access components page; the *TDBGrid* is on the Data Controls page.) These components represent the third table in the one-many-many link. In this example its the*Items* table.

9 To create the link, set properties of these new components as shown in the following table.

Figure 2.6 One-many-many form

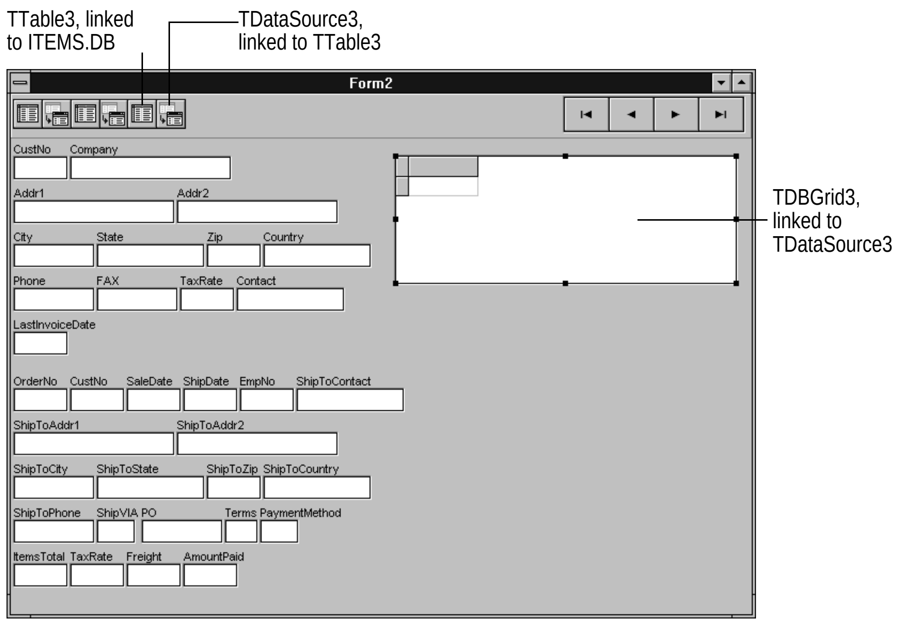


Table 2.7 Important component properties for a one-many-many form

Component	Property	Value	Remarks
TTable (third table)	*Active*	True	False by default. Set *Active* to True after setting all other properties to display data in linked controls.
	DataBaseName	MAST	An alias that specifies where to find the table. Use aliases, not hard-coded paths, to make an application portable and easier to upsize.
	IndexFieldNames	OrderNo	Specifies a column to use to order records in the table. *ByOrderNo* is a secondary index based on *OrderNo*. You can also set the *IndexFields* property to *ByOrderNo*.
	MasterFields	OrderNo	A list of one or more master table fields to link to. Use a semicolon to separate field names in the list.
	MasterSource	DataSource2 (by default)	Identifies a *TDataSource* component linked to the master table.
	Name	Table3 (by default).	Use the Object Inspector to rename it (for example, Items).
	TableName	ITEMS.DB	The name of the third table in the link.
TDataSource (third table)	*DataSet*	Table3 (by default)	Specifies which *TTable* is supplying the data to this component.
	Name	DataSource3 (by default)	Use the Object Inspector to rename it (for example, ItemsSource).
TDBGrid (third table)	*DataSource*	DataSource3 (by default)	This data source provides a restricted view of the data in the *Items* table, based on the values of the fields linked to the *Orders* table.

How it works

A one-many-many form links data from three tables. The first table in the link (the *Customer* table) is the master table. The second table (*Orders*) does double duty: it's the detail table for the first table and the master table for the third table. The third table (*Items*) is a detail table for the second table. In this example, you can link these tables quickly and easily by setting properties because secondary indexes were specified for the tables when they were created.

For more information

For more information about

- Using the Database Form Expert, see page 21.
- Creating tables and adding secondary indexes, see Appendix A, "Using Database Desktop."
- Linking tables, see step 6 on page 25.

Working with fields

Examples in this section show how use code to control field values and display attributes. Delphi's data-aware controls enable end users to view and edit table data, but to access the underlying data or control how it is displayed, you want to create field components. Use the Fields Editor to define a list of fields and work with invisible components of type *TField*.

Creating Tfield components

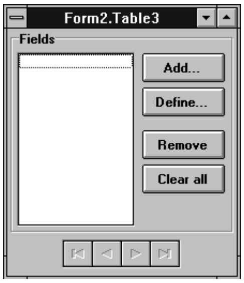
This example explains how to use the Fields Editor to define a list of some or all of the fields (columns) in a table. For each field, Delphi creates a corresponding *TField* component. *TField* components are invisible components that provide access to field values and display attributes. The following example shows how to use the Fields Editor to make a grid display four fields selected from the *Customer* table. It also shows how to specify the field order.

What to do

1. Use the Database Form Expert to build a single-table form that displays all the fields of CUSTOMER.DB in a grid. For detailed instructions, see page 21.
2. To make the grid display data at design time, use the Object Inspector to change the *TTable* component's *Active* property to True.

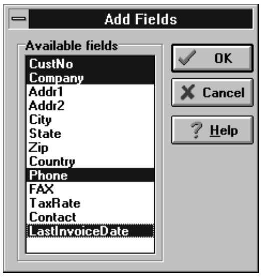
3 Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty, as shown in the following figure:

Figure 2.7 The Fields Editor



4 Click Add to open a dialog box listing the fields in the *Customer* table. By default, all fields are selected. Click CustNo to select it, then control-click to select the Company, Phone, and LastInvoiceDate fields, then click OK to confirm your choices and close the dialog box. In the form, the grid changes: instead of displaying all fields, it displays the only the fields you selected.

Figure 2.8 Adding fields to a data set



5 Use the Fields Editor to change the field order as follows: Click LastInvoiceDate in the list of fields, then drag it to the third place in the list, between Company and Phone. In the form, the grid changes to display columns in their new order.

6 Close the Fields Editor by choosing Close from the Control menu.

7 Press *F9* to run the form. The grid displays the four fields in the order you specified.

How it works

This form is the basis for the full-featured CUSTORD.DFM form in the MASTAPP application. By choosing fields in the Fields Editor, you can tell a *TTable* component which fields to make available to the components that are linked to it. In effect, the Fields Editor changes the logical structure of the table. The Fields Editor also adds *TField* objects to the unit's **type** section (for example, `CustCustNo: TFloatField;`). More accurately, it adds a descendant of the *TField* type appropriate for the data type of the field. For example, when you add the CustNo field to the data set, Delphi adds a *TFloatField* object; when you add the Company field, Delphi adds a *TStringField*, and so on. This tutorial uses the general term *TField* when the specific data type is unimportant.

Note Specifying a list of fields in the Database Form Expert is not the same as using the Fields Editor to define a dataset. The expert places components and controls in a form to create an initial layout, but you must use the Fields Editor to specify the fields in a data set. Use *TField* components, not visual controls, to access fields programmatically.

Understanding the relationship between a *TTable* component, *TField* components, and data-aware controls is crucial to building database applications with Delphi.

For more information

For more information about

- Data sets, the Fields Editor, and *TField* components, see "Using TFields and the Fields Editor" on page 79.
- Using the Database Form Expert, see page 21.

Setting Tfield properties at design time

Because they are invisible, the only way to set the properties of *TField* components at design time is by using the Fields Editor. The following example shows how to set the properties of a *TField* component at design time.

What to do

1. Use the Database Form Expert to build a single-table form that displays all the fields of CUSTOMER.DB in a grid. For detailed instructions, see page 21. (If you're working through these examples in sequence, you can use the form from the previous example and skip to step 5.)
2. To make the grid display data at design time, use the Object Inspector to change the *TTable* component's *Active* property to True. This step is optional: it lets you see what happens to the form as you use the Fields Editor.
3. Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty.
4. Click Add to open a dialog box listing the fields in the *Customer* table. By default, all fields are selected. Control-click to select only the CustNo, Company, Phone, and LastInvoiceDate fields, then click OK to confirm your choices and close the dialog box. In the form, the grid changes: instead of displaying all fields, it displays the only the fields you selected.
5. Click the CustNo field in the Fields Editor's field list and view the properties and values displayed in the Object Inspector. (To open the Object Inspector, choose View | Object Inspector.) Notice that the component's *Name* property is Table1CustNo. Delphi generates this name automatically by appending the field name to the name of the associated *TTable* component. Use this name to refer to the *TField* component in code.

6 Change the *Align* property from *taRightJustify* (the default) to *taCenter*. In the form, the grid changes: values in the CustNo column are centered.

Figure 2.9 TField component properties

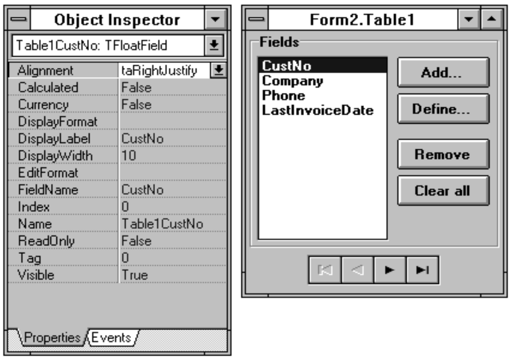


Table1CustNo: TFloatField

Alignment	taRightJustify
Calculated	False
Currency	False
DisplayFormat	
DisplayLabel	CustNo
DisplayWidth	10
EditFormat	
FieldName	CustNo
Index	0
Name	Table1CustNo
ReadOnly	False
Tag	0
Visible	True

CustNo
Company
Phone
LastInvoiceDate

To display a TField component's properties in the Object Inspector, choose a field name from the list in the Fields Editor.

Delphi generates a TField component's name by appending the field name to the name of the associated TTable component. Use this name to refer to the TField in code.

How it works

When you select a field listed in the Fields Editor, you can use the Object Inspector to set an invisible *TField* component's properties, just as you would a visible control. Note that a field's display attributes are properties of the *TField* component, not of the control that displays the value! You only have to set a property once for the *TField* component. Linked controls use the settings automatically. This is a guiding principle of Delphi database applications: use *TField* components to work with database fields.

The following table lists important *TField* design-time properties.

Table 2.8 Important TField design-time properties

Property	Remarks
Alignment	Specifies how to display the field value: left-justified, right-justified, or centered.
Calculated	When True, this field isn't stored in the table but instead is calculated, record by record, by Object Pascal code. Write this code in the table's *OnCalcFields* event handler.
DisplayLabel	The string used by grids as the column header for the corresponding field.
DisplayWidth	The number of characters that a grid column uses to display this field in a grid.
DisplayFormat and *EditMask*	Provides control over input characters for some fields types as they are displayed and edited, for example, in working with commonly formatted values such as dates and telephone numbers.
FieldName	The name of the field in the underlying table. Example: CustNo.
Index	The field's logical position in the data set (first position = 0).
Name	Formed by appending the field name to the name of the *TTable* component. Example: Given a *TTable* name of Customer and a field name of CustNo, the *TField*'s name is CustomerCustNo. This name is used for programmatic access to field's properties (for example: CustomerCustNo.Value).
ReadOnly	When *ReadOnly* is True, data in the corresponding field cannot be modified.
Visible	When True, the corresponding field appears in grids linked to this *TField*.

For more information

For more information about

- Setting a *TField* component's properties, see page 79.
- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.

Reading field values

Every line in the Fields Editor's list of fields represents a Delphi component type *TField*. When you add a field to the list in the Fields Editor, you add a new component to the form (the code appears in the unit's **type** section). Delphi provides *TField* descendents for every available field type. For example, in the *Customer* table, CustNo is of type *TFloatField*, and Company is of type *TStringField*. This example shows how to build a form so you can read field values at run time. You get a field value by reading the *Value* property of the corresponding *TField* component.

What to do

1. Use the Database Form Expert to build a single-table form that displays all the fields of CUSTOMER.DB in a grid. For detailed instructions, see page 21.
2. Use the Object Inspector to change the form's *Name* property to TutorialForm.
3. Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty.
4. Click Add to open a dialog box listing the fields in the *Customer* table. By default, all fields are selected. Control-click to select only the CustNo, Company, Phone, and LastInvoiceDate fields, then click OK to confirm your choices and close the dialog box. The Fields Editor remains open.
5. Place a button control and a standard edit box anywhere in the panel that contains the *TDBNavigator* control (you may need to resize the panel), then set the edit box *Name* property to OutputField.
6. Attach the following code to the button's *OnClick* event:

```
procedure TTutorialForm.Button1Click(Sender: TObject);
begin
  OutputField.Text := Table1Company.Value;
end;
```

7. Press *F9* to run the form. Click a field in the grid, then click the button. The text in the edit control displays the value of the Company field for the current record.

How it works

The code in step 7 does a simple assignment: the edit control's *Text* property gets the value of the *TField* component named Table1Company, where the *TField*'s value is the value of the corresponding field in the current record of the table.

Note In code, refer to a *TField* by its component name, don't use the name of the field in the table. For example, use Table1Company, not Company.

You can assign the *TField* component's value directly to the *Text* property because Table1Company is of type *TStringField*. An edit box's *Text* property and a *TStringField*'s

Value property are of compatible types, so you don't need to do a conversion. However, the following code generates a type mismatch error at compile time, because Table1CustNo is of type *TFloatField*.

```
OutputField.Text := Table1CustNo.Value; {Causes a type mismatch error.}
```

To display a numeric field's value in an edit box, convert the value as follows.

```
OutputField.Text := Table1CustNo.AsString;
```

This code uses the *TField* property *AsString* to read the field value and convert it to a string before assigning it to the edit box's *Text* property. *TField* components have the following properties for converting values: *AsBoolean*, *AsDateTime*, *AsFloat*, *AsInteger*, and *AsString*.

The following code shows examples of how to read field values, display them in edit boxes, and assign them to variables.

```
var
  CustNoDouble: Double;
  CustNoInt: Integer;
  CustNoString: String;
begin
  {Display field value in edit control.}
  OutputField.Text := Table1Company.Value; {Compatible types, no conversion.}
  OutputField.Text := Table1CustNo.AsString; {Convert field value to compatible type.}

  {Assign field value to variables.}
  CustNoDouble := Table1CustNo.Value; {Compatible types, no conversion.}
  CustNoInt := Table1CustNo.AsInteger; {Convert field value to compatible type.}
  CustNoString := Table1CustNo.AsString; {Convert field value to compatible type.}
end;
```

For more information

For more information about

- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.
- Setting a *TField* component's properties, see page 79.

Assigning values to fields

This example shows how to build a form so you can assign field values at run time. You write to a field by assigning a value to the corresponding *TField* component.

What to do

1 Use the Database Form Expert to build a single-table form that displays all the fields of CUSTOMER.DB in a grid. For detailed instructions, see page 21. (If you're working through these examples in sequence, you can use the form from the previous example and skip to step 5.)

2 Use the Object Inspector to change the form's *Name* property to TutorialForm.

3 Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty.

4 Click Add to open a dialog box listing the fields in the *Customer* table. By default, all fields are selected. Control-click to select only the CustNo, Company, Phone, and LastInvoiceDate fields, then click OK to confirm your choices and close the dialog box.

5 Place a button control and a standard edit box anywhere in the panel that contains the *TDBNavigator* control (you may need to resize the panel).

6 Attach the following code to the button's *OnClick* event to change a company's name in the CUSTOMER table based on the current value in the standard edit box:

```
procedure TTutorialForm.Button1Click(Sender: TObject);
begin
  Table1.Edit;
   Table1Company.Value := Edit1.Text;
   Table1.Post;
end;
```

7 Press *F9* to run the form. Type a value into the edit box, then click the button and notice the value of the Company field in the current record in the grid. It should be the same as the text in the edit box.

How it works

The code in step 6 does three things: it puts the table into Edit state, assigns a value to the Company field of the current record, and posts the modified record back to the table (which takes the table out of Edit state). As this example shows, there is a difference between editing field values interactively using controls and editing field values in code. By default, the *AutoEdit* property of a *TDataSource* component is set to True. A user can type into a data-aware control to that data source and modify the value immediately. When the user moves off that record, changes are posted automatically. But, to modify *TField* component values based on values entered in a standard edit control, or to perform table modifications in code, you need to explicitly switch to Edit state, set the value, and post changes.

Note You can safely call *Edit* even if you're already in Edit or Insert state. In such cases, calls to *Edit* have no effect.

The *TField* component and the value you assign must be compatible. In the example, the edit control's *Text* property is compatible with Table1Company, a *TStringField.* However, the following code generates a type mismatch error at compile time, because Table1CustNo is a *TFloatField*.

```
Table1CustNo.Value := OutputField.Text; {Causes a type mismatch error.}
```

To assign the text of an edit control to a numeric field, convert the text as follows.

```
Table1CustNo.AsString := OutputField.Text;
```

This code uses the *TField* property *AsString* to convert the text before assigning it toTable1CustNo. *TField* components have the following properties for converting values: *AsBoolean*, *AsDateTime*, *AsFloat*, *AsInteger*, and *AsString*.

The following code shows examples of how to assign field values, converting them if necessary.

```
Table1CustNo.Value := 12340;
Table1CustNo.AsInteger := 4321;
Table1CustNo.AsString := '5678';

Table2CustNo.Value := Table1CustNo.Value; {Assign value of one TField to another.}
Table2.Fields[0] := Table1.Fields[0]; {Also assign value of one TField to another.}
```

For more information

For more information about

- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.
- Setting a *TField* component's properties, see page 79.

Defining a calculated field

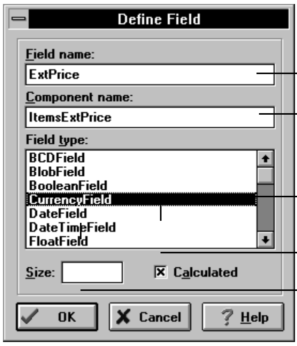
This example shows how to use the Fields Editor and *TField* components to define a calculated field. It calculates the extended price of an item, based on the unit price, quantity ordered, and discount values in the *Items* table. For more information about defining calculated fields, see page 82.

What to do

1. Use the Database Form Expert to build a single-table form that displays all the fields of ITEMS.DB in a grid. For detailed instructions, see page 21.
2. Use the Object Inspector to change the form's *Name* property to OrderForm, change the *TTable* component's *Name* property to Items, and change the *TTable* component's *Active* property to True.
3. In this form's *CreateForm* procedure, change Table1 to Items.
4. Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty.
5. Click Add to open a dialog box listing the fields in the *Items* table. By default, all fields are selected. Click OK to add all the fields to the data set and close the dialog box. In the form, the grid displays a column for each field in the data set.

6 Click Define to open the Define Field dialog box, then enter specifications for a calculated field named ExtPrice as shown in the following figure:

Figure 2.10 Defining a calculated field



Type the field name here.

Delphi automatically creates a name for the TField component by combining the name you type with the name of the TTable component.

Choose a data type from this list.

Check Calculated to specify a calculated field.

Enter the number of characters to display (leave it blank for this tutorial example).

7 Click OK to accept values and close the dialog box. In the form, an empty ExtPrice column appears in the grid. You may have to scroll the grid to the right to see it.

8 Click Define again, then enter specifications for a second calculated field named ItemsSellPrice. It, too, should be given a CurrencyField data type. Click OK.

9 Double click the *OnCalcFields* event of the *TTable* component to open the code window, then enter the following code.

```
procedure TOrderForm.ItemsCalcFields(DataSet: TDataSet);
begin
   if Parts.FindKey([ItemsPartNo]) then
   begin
      ItemsDescription.Value := PartsDescription.Value;
      ItemsSellPrice.Value := PartsListPrice.Value;
   end;
   ItemsExtPrice.Value := ItemsQty.Value *
   ItemsSellPrice.Value * (100 - ItemsDiscount.Value) / 100;
end;
```

10 Type *F9* to run the form. The ExtPrice column fills with calculated values. A calculated field does not display values at design time.

How it works

This form is the basis for the full-featured EDORDERS.DFM form in MASTAPP. It demonstrates the two key aspects of creating a calculated field in Delphi: adding the calculated field to a *TTable* component's data set, and writing code to handle the *TTable* component's *OnCalcFields* event. Delphi sends an *OnCalcFields* event each time the cursor for that table changes. The calculation accesses data through the *TField* components created in the Fields Editor, not the controls placed in the form.

The example attaches code to the *OnCalcFields* event to update the value of the calculated field. You can use the *OnCalcFields* event for other purposes; for example, to do a lookup into another table, perform a complex calculation, or do real-time data acquisition. However, in an *OnCalcFields* event, you can only assign values to calculated fields.

For more information

For more information about

- Programming calculated fields, see page 82.
- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.
- Setting a *TField* component's properties, see page 79.

Formatting field values at design time

This example shows how to format a field value by setting the *DisplayFormat* property at design time.

What to do

1. Use the Database Form Expert to build a single-table form that displays all the fields of CUSTOMER.DB. Specify a vertical layout and labels aligned left. For detailed instructions, see page 21.
2. Use the Object Inspector to change the form's *Name* property to EdCustForm, change the *TTable* component's *Name* property to Cust, and change the *TTable* component's *Active* property to True. In the form, notice the Tax Rate field's display format (example: 8.5).
3. In this form's *CreateForm* procedure, change Table1 to Cust.
4. Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty.
5. Click Add to open a dialog box listing the fields in the *Customer* table. By default, all fields are selected. Click OK to add all the fields to the data set and close the dialog box.
6. In the Field Editor's list of fields, choose Tax Rate, then use the Object Inspector to view the properties of the *TFloatField* named CustTaxRate. The *DisplayFormat* property is blank.
7. In the Object Inspector, enter the following value for the *DisplayFormat* property of CustTaxRate: 0.00%. In the form, the Tax Rate field's display format changes (example: 8.50%).

How it works

Delphi provides several properties that specify a field's display format. The following table lists some of the most frequently-used properties. Any data-aware control linked to a *TField* component uses the *TField*'s display properties.

Table 2.9 Important TField design-time properties

Property	Remarks
Alignment	Displays field value left justified, right justified, or centered within a data-aware control.
Currency	True, numeric field displays monetary values. False, numeric field does not display monetary values.
DisplayFormat	Specifies the format of data displayed in a data-aware component. For more information about valid patterns, see the online help.
DisplayWidth	The number of characters that a grid column uses to display this field in a grid.
DisplayFormat and *EditMask*	Provides control over input characters for some fields types as they are displayed and edited; for example, in working with commonly formatted values such as dates and telephone numbers.
FieldName	The name of the field in the underlying table; for example, CustNo.
Index	The field's logical position in the data set (first position = 0).
Name	Formed by appending the field name to the name of the *TTable* component. Example: Given a *TTable* name of Customer and a field name of CustNo, the *TField*'s name is CustomerCustNo. This name is used for programmatic access to field's properties (for example, CustomerCustNo.Value).
Visible	When True, the corresponding field can appear in linked grids.

For more information

For more information about

- Formatting field values, see page 83.
- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.
- Setting a *TField* component's properties, see page 79.

Formatting field values at run time

This example shows how to format a field value by setting the *DisplayText* property in code.

What to do

1. Use the Database Form Expert to build a single-table form that displays all the fields of CUSTOMER.DB. (If you're working through these examples in sequence, you can use the form from the previous example and skip to step 6.) Specify a vertical layout and labels aligned left. For detailed instructions, see page 21.
2. Use the Object Inspector to change the form's *Name* property to EdCustForm, change the *TTable* component's *Name* property to Cust, and change the *TTable* component's *Active* property to True. In the form, notice the Phone field displays U.S phone numbers (example: 808-555-0269).
3. In this form's *CreateForm* procedure, change Table1 to Cust.

4 Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty.

5 Click Add to open a dialog box listing the fields in the *Customer* table. By default, all fields are selected. Click OK to add all the fields to the data set and close the dialog box.

6 In the Fields Editor's list of fields, choose Phone, then use the Object Inspector to view the events of the *TStringField* named CustPhone.

7 Double-click the *OnGetText* event to open the code window, then add code to handle the event as follows.

```
procedure TEdCustForm.CustPhoneGetText(Sender: TField;
   var Text: OpenString; DisplayText: Boolean);
begin
      if DisplayText then
      begin
            Text := CustPhone.Value;
            Delete(Text, 4, 1);
            Insert('(', Text, 1);
            Insert(')', Text, 5);
      end;
end;
```

In this sample code, *DisplayText* is True, meaning you're not actually editing the field in question, but modifying the display of the field. When *DisplayText* is True, characters that would be invalid for the user to enter, such as parentheses in this case, can be inserted into the field for display purposes only.

8 Press *F9* to run the form. The Phone field's display format changes (example: (808)555-0269).

How it works

Use the *OnGetText* event to format field values. Delphi calls a *TField* component's *OnGetText* method whenever it's about to display the value of a field onscreen, for example, when redrawing a data-aware component. Delphi ignores a *TField* component's *DisplayFormat* property when you add code to its *OnGetText* event.

In this example, *CustPhoneGetText* copies the phone number string to variable *Text*, then uses the *Insert* and *Delete* procedures to place parentheses around the first three digits. This gives U.S. phone numbers a more traditional format: (713) 555-1212, instead of 713-555-1212 as stored in the file. The *DisplayText* property is True when a *TField* is displaying data but is not available for editing. When a *TField* is in Edit state, *DisplayText* is *False*.

Note *DisplayFormat* and *EditMask* are ignored if an *OnGetText* event handler exists.

For more information

For more information about

- Formatting fields, see page 85.
- Editing display properties, see page 83.

- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.
- Setting a *TField* component's properties, see page 79.

Searching for field values

This example shows how to search for values in keyed (indexed) fields in a table. To search for values in unkeyed fields, use a query. See "Using queries and ranges" on page 47 for more information.

What to do

This example shows how to display a record in the Customer table by searching for a customer number entered by the user.

1. Use the Database Form Expert to build a single-table form that displays fields from CUSTOMER.DB. Specify a vertical layout and labels aligned left. For detailed instructions, see page 21.
2. Place a button control in the panel that contains the *TDBNavigator* control.
3. Double-click the button to open a code window, then add code to handle the button's *OnClick* event as follows.

```
procedure TForm2.Button1Click(Sender: TObject);
var UserCustNo: String;
begin
  UserCustNo := InputBox('Search', 'Enter a Customer Number:', ' ');
   if not Table1.FindKey([UserCustNo]) then
  MessageDlg('Not found.', mtInformation, [mbOK], 0);
end;
```

4. Add the *Dialogs* unit to the **uses** clause of your form unit. The *InputBox* function used in the previous step resides in the *Dialogs* unit.
5. Press *F9* to run the form. Click the button and enter a number in the input dialog box (try 1560). If that number is a customer number in the table, Delphi displays the corresponding record. Otherwise, it displays a dialog box.

How it works

FindKey takes an array, where each array value represents a value to search for in the corresponding key field in the table. *FindKey* searches for the first array value in the first key field, the second array value in the second key field, and so on. In this example, the array contains one value, so *FindKey* searches for it in the first key field of the table, which is CustNo.

The following example shows how to search for the name Frank P. Borland in a table where the key fields are LastName, FirstName, and MiddleInital.

```
begin
     if not Table1.FindKey(['Borland', 'Frank', 'P.']) then
        MessageDlg('Not found.', mtInformation, [mbOK], 0);
end;
```

For more information

For more information about

- Searching for field values, see page 70.
- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.

Validating data entry

This section describes ways to make sure the user doesn't enter invalid data into tables. There are three different approaches to validating data: table-based, field-based, and record-based.

Most forms use a combination of validation techniques.

- Table-based validation. Build in validity checks when you create tables. Using the Database Desktop, you can specify a wide range of validity checks, including minimum and maximum values, formatting pictures, and referential integrity, all without writing a line of code. Table-based validation rules are enforced by the database when the data is posted. Multiple exceptions will be raised, one after another, until all fields match the validation criteria. Delphi enforces some table-based validation rules before posting (for example, whether a field is required or not).

  For more information about the DBD, see Appendix A, "Using Database Desktop."

- Field-based validation. There are two methods:

  Create an *OnValidate* event handler. The handler is called when the field's value is modified, whether by user input or when the field is assigned a value in code.

  For fields that must get values from a user (for example, a password or part number) set the *TField*'s *Required* property to True, and write an *OnValidate* event handler for the field. Before a record is posted, an exception occurs for any fields that have a **nil** value.

- Record-based validation. Use this approach when other fields are involved in determining if a field is valid. For example, if it is invalid to use a credit card for purchases under $10, you want to give a user the chance to enter both a payment method and an invoice amount before performing validation on the payment method. Otherwise, since a new order starts with a zero invoice amount, you would require the entry of all line items before permitting specification of a payment method.

  Record-based validation should be performed in a table's *BeforePost* event handler. Raising an exception there prevents posting from occurring. Use the *TField FocusControl* method to inform the user which field is invalid. For an example of this technique, see "Writing code to check field values" on page 46.

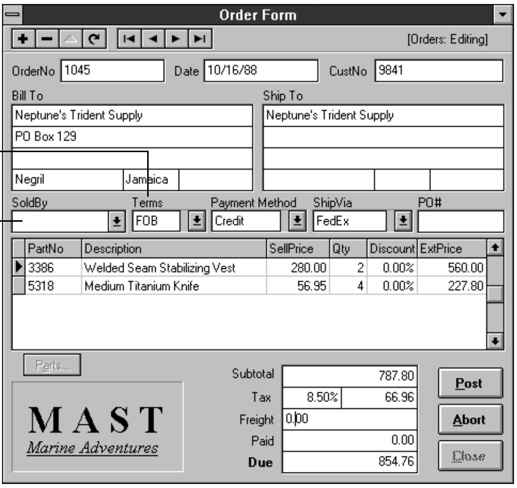
Using lists and lookups

The following examples show two techniques for using lists to restrict user input. The first example presents a list of items entered by hand. The second example presents a lookup filled with items from a field (column) of a table.

Figure 2.11 Lists and lookups

The Terms field is a *TDBComboBox* control. The list items are hard-coded. See page 44.

The SoldBy field is a *TDBLookupCombo* control. It reads list items from a field (column) of a table. See page 45.



Order Form

[Orders: Editing]

OrderNo	1045	Date	10/16/88	CustNo	9841

Bill To	Ship To
Neptune's Trident Supply	Neptune's Trident Supply
PO Box 129	
Negril Jamaica	

SoldBy	Terms	Payment Method	ShipVia	PO#
	FOB	Credit	FedEx	

PartNo	Description	SellPrice	Qty	Discount	ExtPrice
3386	Welded Seam Stabilizing Vest	280.00	2	0.00%	560.00
5318	Medium Titanium Knife	56.95	4	0.00%	227.80

Subtotal		787.80
Tax	8.50%	66.96
Freight	0.00	
Paid		0.00
Due		854.76

What to do (list)

The following steps describe how to create the drop-down list labeled Terms in the Edorders form in MASTAPP. The Terms field specifies the payment terms for an order. Allowed values are Prepaid, Net 30, and COD.

1 Open the MASTAPP project, then open the Edorders form.

2 Place a *TTable* component. Set its *DatabaseName* and *TableName* properties to link it to the *Orders* table. Change its *Name* property to Orders.

3 Place a *TDataSource* component. Set its *DataSet* property to the Orders *TTable*. Change its *Name* property to OrdersSource.

4 Place a *TDBComboBox* control. Set its properties as shown in the following table.

Table 2.10 Important TDBComboBox properties

Property	Value	Remarks
DataField	Terms	The name of the field in the table this control is linked to.
DataSource	OrdersSource	The *TDataSource* component this control is linked to.
Items	Prepaid Net 30 COD	Each list item is on its own line. In other words, when you enter list items, press *Enter* after each item.

How it works

A *TDBComboBox* control works like a standard combo box control, but it's linked to a field in a table. Its *Items* property stores the list items. When the user chooses an item from the list, that value is assigned to the corresponding data field in the current record. You can add list items by hand at design time or specify them in code at run time.

For more information

For more information about

- Using lists and combo boxes, see "Using list and combo boxes" on page 109.
- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.

What to do (lookup)

The following steps describe how to create the drop-down lookup list labeled SoldBy in the Edorders form in MASTAPP. The combo box labeled SoldBy displays a list of employee names, but it stores employee numbers. When the user chooses a name from the list, Delphi looks up the corresponding employee number and writes that value to the *Orders* tables.

1. Open the MASTAPP project, then open the Edorders form.
2. Place two *TTable* components. Set the first component's *DatabaseName* and *TableName* properties to link it to the *Orders* table, and change its *Name* property to Orders. Set the second component's *DatabaseName* and *TableName* properties to link it to the *Employee* table, and change its *Name* property to Emps.
3. Place two *TDataSource* components. Set the first component's *DataSet* property to the Orders *TTable* and change its *Name* property to OrdersSource. Set the second component's *DataSet* property to the Emps *TTable* and change its *Name* property to EmpsSource.
4. Place a *TDBLookupCombo* control. Set its properties as shown in the following table.

Table 2.11 Important TDBComboBox properties

Property	Value	Remarks
DataSource	OrdersSource	The *TDataSource* component this control is linked to.
DataField	EmpNo	The name of the field in the table this control is linked to.
LookupSource	EmpsSource	The *TDataSource* component used to identify the table from which to look up field values.
LookupDisplay	FullName	The field (column) this control reads from to display list values to the user. FullName is a calculated field in the table pointed to by LookupSource.
LookupField	EmpNo	The field to search for a value corresponding to the value in the *LookupDisplay* field. This value is assigned to the field specified in *DataField*.

How it works

The *TDBLookupCombo* control dynamically accesses a column from a table (specified in *LookupDisplay*) and displays it to the user. When the user chooses an item, Delphi finds the value in the table specified in *LookupSource*. Then it reads the value of the field specified in *LookupField*. Finally, it assigns that value to the field specified in *DataField* in the table specified in *DataSource*.

In this example, Delphi fills a drop-down box with employee names from the FullName field of the *Employee* table. FullName is a calculated field defined for the *Employee* table. To see how FullName is calculated, select *Employee TTable* component, then examine its *OnCalcFields* event. When the user chooses a name from the list, Delphi reads the EmpNo field to get the employee number for that name. Then it assigns that value to the EmpNo field of the *Orders* table.

For more information

For more information about

- Using lists and combo boxes, see "Using list and combo boxes" on page 109.
- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.

Writing code to check field values

This example shows how to validate a field value after the user has entered it. The following steps show how to add code to the Edorders form in MASTAPP to disallow a sale date later than the current date. The example uses the standard function *Now* to get the current timestamp.

What to do

1 Open the MASTAPP project, then open the Edorders form.

2 Double-click the *TTable* component named Orders to open the Fields Editor.

3 In the Fields Editor's list of fields, click SaleDate, then use the Object Inspector to view the events of the *TDateTimeField* component named OrdersSaleDate.

4 Double-click the *OnValidate* event to open the code window, then add code to handle the *OnValidate* event as follows.

```
procedure TOrderForm.OrdersSaleDateValidate(Sender: TField);
begin
 if OrdersSaleDate.Value > Now then
  raise Exception.Create('Cannot enter a future date');
end;
```

How it works

To check field values after the user enters them, write code to handle the *OnValidate* event of the appropriate *TField* component. The example code raises an exception if the date a user enters is later than today's date. Make sure all fields are either initialized to a valid value in an *OnNewRecord* event, or have their *TField Required* properties set to *True*.

For more information

For more information about

- Working with *TField* components, see page 34.
- Using the Fields Editor, see page 30.

Using queries and ranges

The examples in this section show how to select a set of records from a table. In Delphi, you can do this by issuing SQL statements or by setting a range (filter). SQL statements can be either static or dynamic; that is, they can be fixed or include parameters where values are provided at run time. For more information, see Chapter 5, "Using SQL in applications."

What to do (static query)

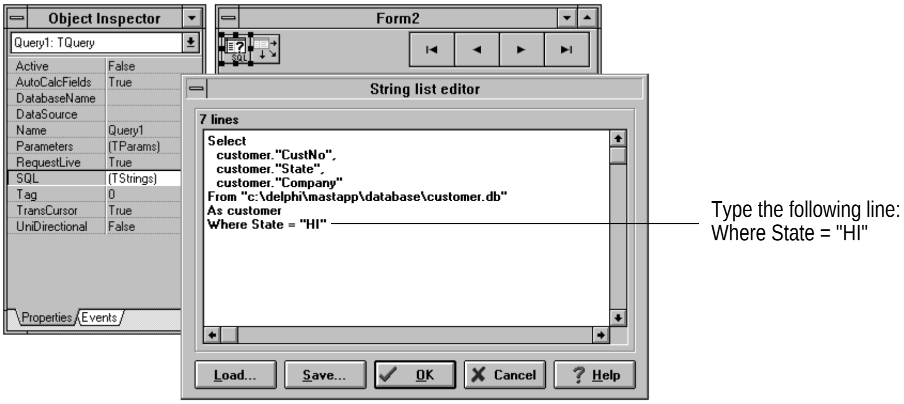
This example shows how to create and execute a static query and display the results in a form.

1. Use Form Expert to build a form based on a query of the *Customer* table (for detailed instructions, see page 21). Use only the CustNo, Company, and State fields (for simplicity). Specify a grid layout.
2. Use the Object Inspector to set the *TQuery* component's *Active* property to True. The grid displays the query results (by default, the query selects all records).
3. Set the *TQuery* component's *Active* property to False. The grid empties. (*Active* must be False to change the query.)
4. Use the Object Inspector to display the *TQuery* component's *SQL* property, and type the following statement after the last line in the SQL statement in the String List Editor window:

   where State = "HI"
5. Click OK to close the String List Editor.

6 Set the *TQuery* component's *Active* property to True. The grid displays only those records where the value of the State field is "HI." (Optional: Press *F9* to run the form.)

Figure 2.12 Setting a TQuery's SQL property



Active	False
AutoCalcFields	True
DatabaseName	
DataSource	
Name	Query1
Parameters	(TParams)
RequestLive	True
SQL	(TStrings)
Tag	0
TransCursor	True
UniDirectional	False

```
Select
  customer."CustNo",
  customer."State",
  customer."Company"
From "c:\delphi\mastapp\database\customer.db"
As customer
Where State = "HI"
```

Type the following line:
Where State = "HI"

How it works

Delphi reads the SQL statements assigned to the *TQuery* component's *SQL* property and passes them to the server without any intermediate interpretation. (Use double quotes in SQL statements, not single quotes as in Pascal code.) The server executes the query and returns the results to Delphi, and Delphi displays the result set in the grid.

Note: By default, a *TQuery* component's *RequestLive* property is set to False. When *RequestLive* is False, the result set returned by a query is read-only. Users cannot modify data in the form. If you want users to be able to update data returned by a query of a single table, then you can set *RequestLive* to True to request a live result set. If the query is updateable, *CanModify* returns *True*.

For more information

For more information about

- Using *TQuery* components and SQL in applications, see Chapter 5.
- *RequestLive* behavior and restrictions, see Chapter 5, "Using SQL in applications."
- Using the Database Form Expert, see page 21.

What to do (dynamic query)

This example shows how to create and execute a dynamic query and display the results in a form. A dynamic query consists of SQL statements that include parameters (variables) whose values can be assigned at design time or run time. Steps 1 through 6 show how to set parameters at design time. Steps 7 through 9 show how to do it at run time.

1 Use Form Expert to build a form based on a query of the *Customer* table (for detailed instructions, see page 21). Use only the CustNo, Company, and State fields (for simplicity). Specify a grid layout. If you're working through these examples in sequence, you can use the form from the previous example.

2 Use the Object Inspector to display the *TQuery* component's *SQL* property, and type the following statement after the last line in the SQL statement in the String List Editor window:

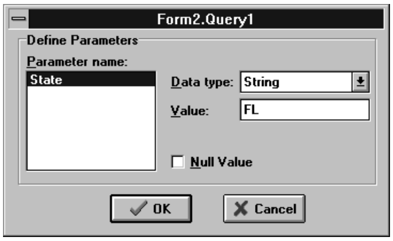
   where State = :State

3 The SQL statement in the previous step contains the field name State and the parameter *State* (a parameter is an arbitrary string preceded by a colon). Click OK to close the String List Editor.

4 Right-click the *TQuery* component, then choose Define Parameters from the pop-up menu. The Define Parameters dialog box opens, ready for you to assign a field type and a value to the parameter *State*.

5 Choose a field type of String, and enter a value of FL, then click OK to close the dialog box.

Figure 2.13 Defining a query parameter


Form2.Query1

Define Parameters

Parameter name: State

Data type: String

Value: FL

☐ Null Value

To define the parameter State, choose a field type of String and enter a value of FL.

6 Set the *TQuery* component's *Active* property to True. The grid displays the query results (all records where State = FL).

7 Place a button control in the panel that contains the *TDBNavigator* control.

8 Double-click the button to open a code window, then write the following code to handle its *OnClick* event.

```
procedure TForm2.Button1Click(Sender: TObject);
begin
    Query1.DisableControls;
      try
         Query1.Active := False;
         Query1.Params[0].AsString := 'HI';
         Query1.Active := True;
      finally
         Query1.EnableControls;
      end;
end;
```

9 Press *F9* to run the form, then click the button. The grid displays the query results (all records where State = HI).

How it works

In a SQL statement, a string preceded by a colon (like :State in the example) represents a parameter. At design time, Delphi recognizes parameters and displays them in the

Define Parameters dialog box. Then at run time, it assigns the specified values to the corresponding parameters.

You can assign values to query parameters at run time using the *Params* property. It stores parameter values in an array, where an index of 0 represents the first parameter, an index of 1 represents the second parameter, and so on. So, this statement assigns the value HI to the first parameter.

```
Query1.Params[0].AsString := 'HI';
```

You need to set the *TQuery* component's *Active* property to False before setting parameter values, then set it to True again to update the query. The calls to *DisableControls* and *EnableControls*, respectively, freeze and restore display capabilities of controls linked to the *TQuery* component. This disables data-aware control while the result set is updated.

For more information

For more information about

- Dynamic SQL statement, see page 121.
- Using SQL in applications, see Chapter 5.
- Using TQuery components, see page 115.
- The SQL property, see page 117.
- Using the Database Form Expert, see page 21.

Setting a range

This example explains how to select a set of records by setting a range (also called a filter). To set a range on a Paradox or dBASE table, work with keyed (indexed) fields. To set a range on a SQL table, you can specify the fields to be used as indexes using the *IndexFieldNames* property.

What to do

1. Use the Database Form Expert to build a single-table form that displays all the fields of CUSTOMER.DB in a grid. For detailed instructions, see page 21.
2. Use the Object Inspector to change the form's *Name* property to CustForm, change the *TTable* component's *Name* property to Cust, and change the *TTable* component's *Active* property to True.
3. In this form's *CreateForm* procedure, change Table1 to Cust.
4. Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty.
5. Click Add to open a list box for the fields in the *Items* table. By default, all fields are selected. Click OK to add all the fields to the data set and close the list box. Close the Fields Editor. In the form, the grid displays a column for each field in the data set.
6. Place a button control in the panel that contains the *TDBNavigator* control.

7 Double-click the button to open a code window, then add code to handle the button's *OnClick* event as follows.

```
procedure CustForm.Button1Click(Sender: TObject);
begin
   Cust.DisableControls;
try
   Cust.SetRangeStart;
   CustCustNo.Value := 3000;
   Cust.KeyExclusive := False;

   Cust.SetRangeEnd;
   CustCustNo.Value := 4000;
   Cust.KeyExclusive := True;

   Cust.ApplyRange;
finally
   Cust.EnableControls;
end;
end;
```

8 Press *F9* to run the form, then click the button. The grid displays records for customer numbers from 3,000 to 3,999.

How it works

This example sets a range to include records for customer numbers from 3,000 to 3,999. Setting a range on a table affects the values displayed in data-aware components linked to that table. That's why the routine in step 7 begins by calling *Cust.DisableControls* and ends by calling *Cust.EnableControls*. *DisableControls* disables the display capabilities of controls linked to the specified table; in effect, it freezes them. *EnableControls* restores the controls to an active state.

Important Always use a **try. . . finally** statement, followed by *EnableControls*. Otherwise, after an exception occurs, data controls are inactive.

The call to *Cust.SetRangeStart* marks the beginning of a code block that sets the minimum value of the range. The next line assigns a value to the *TField* component named *CustCustNo.* (CustNo is the first and only key field in the *Customer* table.) Setting *Cust.KeyExclusive* to False indicates that the value is not to be excluded in the range. So, in this example, a customer number 3,000 would be included in the range.

The call to *Cust.SetRangeEnd* marks the beginning of a code block that sets the maximum value of the range. The next line assigns a value to *CustCustNo*. Setting *Cust.KeyExclusive* to True indicates that the value is to be excluded from the range. A customer number 4, 000 would not be included.

The call to *Cust.ApplyRange* puts the range settings into effect. (To cancel a range, call *CancelRange*.)

For more information

For more information about

- Setting ranges and filtering data, see page 72.
- Using the Fields Editor, see page 30.

Printing reports and forms

This section describes how to use Delphi to print reports created using ReportSmith, and how to print a Delphi form. Designing reports with ReportSmith is described in *Creating Reports*.

Note To run reports, the *TReport* component needs to locate the RS_RUN directory. You can put the RS_RUN directory in your DOS PATH, or you can put an "EXEpath =" entry in the Delphi RS_RUN.INI file.

What to do: printing reports

This example show how to print a ReportSmith report that lists MAST customers.

1 Choose File | New Form to open the Browse Gallery, then choose Blank form and click OK to create a blank form.

2 Place a *TReport* component anywhere in the form. (The *TReport* component is on the Data Access components page.)

3 Use the Object Inspector to set the *TReport* component's properties as shown in the following table.

Table 2.12 Important TReport properties

Property	Value	Remarks
Preview	True	When *Preview* is True, Delphi displays the report onscreen only; when *Preview* is False, Delphi sends the report to the printer.
ReportDir	C:\DELPHI\DEMOS\DB\MASTAPP\REPORTS	The path and directory where the report file resides.
ReportName	CUSTLIST.RPT	The report file name.

4 Place a button control anywhere in the form.

5 Double-click the button to open a code window, then write code to handle its *OnClick* event, as follows.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Report1.Run;
end;
```

6 Press *F9* to run the form.

7 Click the button to run the report. If the *TReport* component's *Preview* property is set to True, Delphi displays the report onscreen; if *Preview* is False, Delphi sends the report to the printer.

How it works

The *Run* method of *TReport* opens the run-time version of ReportSmith, which prints or displays a report as specified by the *TReport* component's properties. When you're designing a form, you can double-click a *TReport* component to open the full version of ReportSmith and build a report.

For more information

For more information about

- Using *TReport* components, see page 88.
- Using ReportSmith, see *Creating Reports*.

What to do: printing forms

You can print a form by calling its *Print* method. Following is the code called by the Print button in the main form (MAIN.DFM) of MASTAPP. It prints whichever Delphi form is displayed "on top," that is, in front of the others.

```
procedure TMainForm.PrintTopForm(Sender: TObject);
var TopForm: TForm;
begin
  TopForm := Screen.Forms[1]; {For a single form app, this should be Screen.Forms[0]!}
  if not TopForm.Visible then ShowMessage('Nothing to print!')
  else if MessageDlg(Format('Print "%s" form?', [TopForm.Caption]),
    mtConfirmation, mbOkCancel, 0) = mrOk then TopForm.Print;
end;
```

How it works

The *Print* method for the *TForm* class prints a form as it appears onscreen.

For more information

For more information about

- The *TForm* class, see the online Help.
- Designing forms, see the *User's Guide*.

Chapter

3

Using data access components and tools

This chapter describes how to use key Delphi features and tools when building database applications, including:

- The *TSession* component.
- Dataset components (*TTable* and *TQuery*), their properties, and their methods.
- *TDataSource* components, their properties, and their methods.
- *TField* objects, their properties, and their methods.
- The Fields Editor to instantiate and control *TField* objects.
- *TReport* and *TBatchMove* components.

This chapter provides an overview and general description of data access components in the context of application development. For in-depth reference information on database components, methods, and properties, see the online VCL reference.

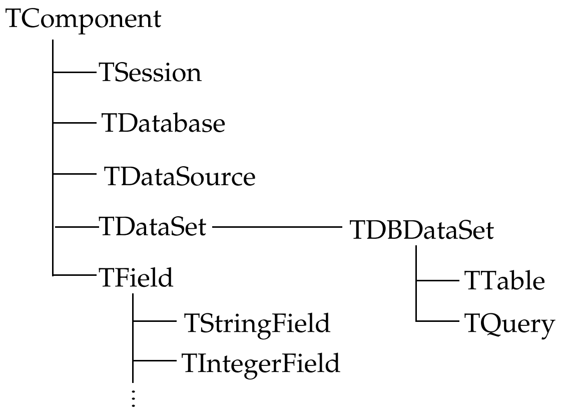
Database components hierarchy

The Delphi database component hierarchy is important to show the properties, methods, and events inherited by components from their ancestors. The most important database components are

- *TSession*, a global component created automatically at run time. It is not visible on forms either at design time or run time.
- *TDatabase*, component that provides an additional level of control over server logins, transaction control, and other database features. It appears on the Data Access component page.
- *TDataSet* and its descendents, *TTable* and *TQuery*, collectively referred to as dataset components. *TTable* and *TQuery* components appear on the Data Access component page.

- *TDataSource*, a conduit between dataset components and data-aware components. It appears on the Data Access component page.
- *TFields*, components corresponding to database columns, created either dynamically by Delphi at run time or at design time with the Fields Editor. Data controls use them to access data from a database. In addition, you can define calculated fields whose values are calculated based on the values of one or more database columns.

Figure 3.1 Delphi Data Access components hierarchy



This chapter describes most of these components and the tools that Delphi provides to work with them. The *TQuery* component is described in Chapter 5, "Using SQL in applications." The *TDatabase* component is described in Chapter 6, "Building a client/server application."

Using the TSession component

The *TSession* component is rarely used, but can be useful for some specialized purposes. Delphi creates a *TSession* component named "Session" each time an application runs. You cannot see nor explicitly create a *TSession* component, but you can use its methods and properties to globally affect the application.

Controlling database connections

TSession provides global control over database connections for an application. The *Databases* property of *TSession* is an array of all the active databases in the session. The *DatabaseCount* property is an integer specifying the number of active databases (*TDatabase* components) in the Session. For more information on *TDatabase*, see Chapter 6, "Building a client/server application."

KeepConnections is a Boolean property that specifies whether to keep inactive database connections. A database connection becomes inactive when a *TDatabase* component has no active datasets. By default, *KeepConnections* is True, and an application will maintain its connection to a database even if the connection is inactive. This is generally preferable if an application will be repeatedly opening and closing tables in the database. If *KeepConnections* is False, a database connection will be closed as soon as the connection is inactive. The *DropConnections* method will drop all inactive database connections.

The *NetFileDir* property specifies the directory path of the BDE network control directory. The *PrivateDir* property specifies the path of the directory in which to store temporary files (for example, files used to process local SQL statements). You should set this property if there will be only one instance of the application running at a time. Otherwise, the temporary files from multiple application instances will interfere with each other.

Getting database information

TSession has a number of methods that enable an application to get database-related information. Each method takes a *TStrings* component as its parameter and returns into a *TStrings* the specified information:

Table 3.1 TSession methods

Method	Returns
GetAliasNames	Defined BDE alias names.
GetAliasParams	Parameters for the specified BDE alias.
GetDatabaseNames	Database names and BDE aliases defined.
GetDriverNames	Names of BDE drivers installed.
GetDriverParams	Parameters for the specified BDE driver.
GetTableNames	All table names in the specified database.

For more information on these methods, see the online *VCL Reference*.

Using datasets

TTable and *TQuery* component classes are descended from *TDataSet* through *TDBDataSet*. These component classes share a number of inherited properties, methods, and events. For this reason, it is convenient to refer to them together as datasets, when the discussion applies to both *TTable* and *TQuery*.

This section describes the features of datasets that are common to *TTable* and *TQuery*. A subsequent section discusses features unique to *TTable*. Chapter 5, “Using SQL in applications” describes features unique to *TQuery*.

Note *TStoredProc* is also a dataset component since it is descended from *TDBDataset*. Therefore, much of this section also applies to *TStoredProc* if the stored procedure returns a result set rather than a singleton result. For more information on *TStoredProc*, see Chapter 6, “Building a client/server application.”

Dataset states

A dataset can be in the following states, also referred to as *modes*:

Table 3.2 Dataset states

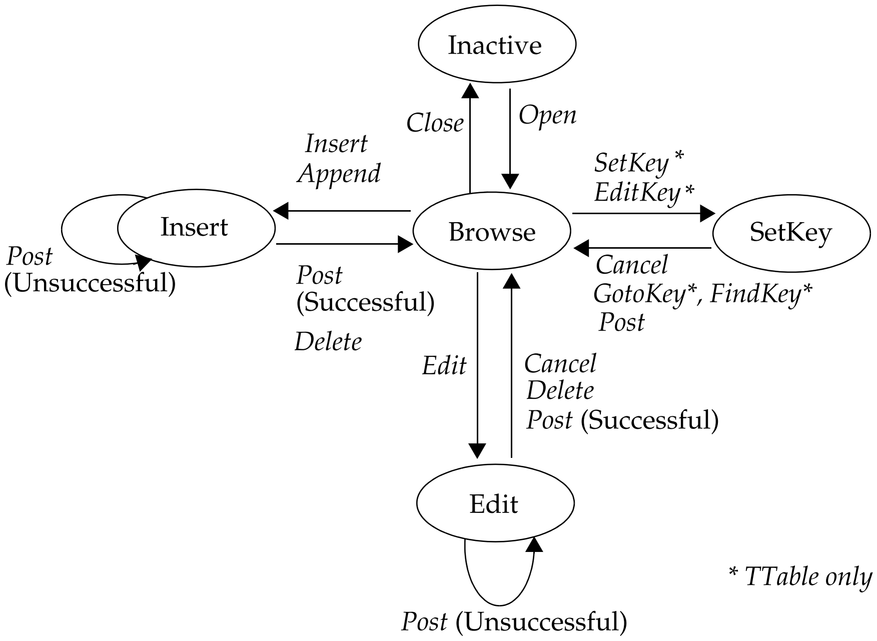
State	Description
Inactive	The dataset is closed.
Browse	The default state when a dataset is opened. Records can be viewed but not changed or inserted.
Edit	Enables the current row to be edited.
Insert	Enables a new row to be inserted. A call to Post inserts a new row.
SetKey	Enables FindKey, GoToKey, and GoToNearest to search for values in database tables. These methods only pertain to TTable components. For TQuery, searching is done with SQL syntax.
CalcFields	Mode when the OnCalcFields event is executed; prevents any changes to fields other than calculated fields. Rarely used explicitly.

An application can put a dataset into most states by calling the method corresponding to the state. For example, an application can put Table1 in Insert state by calling `Table1.Insert` or Edit state by calling `Table1.Edit`. A number of methods return a dataset to Browse state, depending on the result of the method call. A call to *Cancel* will always return a dataset to Browse state.

CalcFields mode is a special case. An application cannot explicitly put a dataset into *CalcFields* mode. A dataset automatically goes into *CalcFields* mode when its *OnCalcFields* event is called. In *OnCalcFields*, an exception will occur if an application attempts to assign values to non-calculated fields. After the completion of *OnCalcFields*, the dataset returns to its previous mode.

The following diagram illustrates the primary dataset states and the methods that cause a dataset to change from one mode to another.

Figure 3.2 Dataset state diagram



The *State* property specifies the current state of a dataset. The possible values correspond to the above states and are *dsInactive*, *dsBrowse*, *dsEdit*, *dsInsert*, *dsSetKey*, and *dsCalcFields*.

The *OnStateChange* event of *TDataSource* is called whenever the state of a data source's dataset changes. For more information, see "Using TDataSource events" on page 78.

Opening and closing datasets

Before an application can access data through a dataset, the dataset must be open. There are two ways to open a dataset:

- Set the dataset's *Active* property to True, either at design time through the Object Inspector, or programmatically at run time. For example,

```
Table1.Active := True;
```

- Call the dataset's *Open* method at run time. For example,

```
Query1.Open
```

Both of these statements open the dataset and put it into Browse state.

Similarly, there are two ways to close a dataset:

- Set the dataset's *Active* property to False, either at design time through the Object Inspector, or programmatically at run time. For example,

```
Query1.Active := False;
```

- Call the dataset's *Close* method. For example,

```
Table1.Close;
```

Both of these statements return a dataset to Inactive state.

Navigating datasets

There are two important concepts in understanding how Delphi handles datasets: *cursors* and *local buffers*. Each active dataset has a cursor, which is essentially a pointer to the *current row* in the dataset. A number of rows of data before and after the cursor are fetched by Delphi into the local buffer. Delphi will always fetch a number of rows into the local buffer sufficient to display the current row, plus an additional number of rows to reduce the refresh time as the user scrolls up or down in the dataset:

Table 3.3 Navigational methods and properties

Method or property	Description
First method	Moves the cursor to the first row of a dataset.
Last method	Moves the cursor to the last row of the dataset.
Next method	Moves the cursor to the next row in the dataset.
Prior method	Moves the cursor to the prior row in the dataset.
BOF property	True when cursor is known to be at beginning of dataset, otherwise False.
EOF property	True when cursor is known to be at end of dataset, otherwise False.
MoveBy(n) method	Moves the cursor *n* rows forward in dataset, where *n* is a positive or negative integer.

Many of these methods are encapsulated in the *TDBNavigator* component. For more information on *TDBNavigator*, see Chapter 4, "Using Data Controls."

The Next and Prior methods

The *Next* method moves the cursor down (forward) by one row in the table. For example, the following code for a button's *OnClick* event moves to the next row in Table1:

```
Table1.Next;
```

Similarly, the *Prior* method moves the cursor up (backward) by one row in the dataset. For example, to move to the previous row in the table, a button's *OnClick* text could be:

```
Table1.Prior
```

The First and Last methods

As their names imply, the *First* and *Last* methods move to a dataset's first and last rows, respectively. For example, the following code for a button's *OnClick* event moves the cursor to the first row in Table1:

```
Table 1.First;
```

Similarly, the *Last* method moves to the last row in the dataset. To move to the last row in the table, a button's *OnClick* text could be:

```
Table1.Last
```

The BOF and EOF properties

BOF is a read-only Boolean property that indicates whether a dataset is known to be at its first row. The *BOF* property returns a value of True only after:

- An application first opens a table
- A call to a Table's *First* method
- A call to a Table's *Prior* method fails

Databases are dynamic; while one application is viewing data, another may be inserting rows before or after the first application's notion of the current row. For this reason, for a non-empty table, it is unsafe to assume *BOF* is True.

For example, consider the following code:

```
Table1.Open; {BOF = True}
Table1.Next; {BOF = False}
Table1.Prior; {BOF = False}
```

After this code executes, *BOF* is False, even if there are no records before the current row. Once the table is open, Delphi can only determine *BOF* when an application explicitly calls *First* or a call to *Prior* fails. Similarly, Delphi can only determine *EOF* when an application explicitly calls *Last* or a call to *Next* fails.

The following code sample demonstrates a common technique for using the *BOF* property:

```
while not Table1.BOF do
begin
   DoSomething;
   Table1.Prior;
end;
```

In this code sample, the hypothetical function *DoSomething* is called on the current record and then on all the records between the current record and the beginning of the dataset. The loop will continue until a call to *Prior* fails to move the current record back. At that point, *BOF* will return a value of True and the program will break out of the loop.

To improve performance during the iteration through the table, call the *DisableControls* method before beginning the loop. This prevents data controls from displaying the iteration through the table, and speeds up the loop. After the loop completes, call the *EnableControls* method. Make sure to use a **try...finally...end** statement with the call to *EnableControls* in the **finally** clause. Otherwise, an exception will leave the application's controls inactive.

The same principles apply to the *EOF* property, which returns a value of True after:

- An application opens an empty dataset
- A call to a Table's *Last* method
- A call to a Table's *Next* fails

The following code sample provides a simple means of iterating over all the records in a dataset:

```
Table1.DisableControls;
try
   Table1.First;
while not Table1.EOF do
begin
   DoSomething;
   Table1.Next;
end;
finally
   Table1.EnableControls;
end.
```

In this case, the *Next* method and the *EOF* property are used together to reach the end of the dataset.

Caution A common error in using such properties in navigating a dataset is to use a **repeat. . . until** loop while forgetting to call *Table1.Next*, as in the following example:

```
Table1.First;
repeat
   DoSomething;
until Table1.EOF;
```

If code like this were executed, the application would appear to "freeze," since the same action would be endlessly performed on the first record of the dataset, and the *EOF* property would never return a value of True.

On an empty table, opening or executing any navigational methods will return True for both *BOF* and *EOF*.

The MoveBy function

The *MoveBy* function enables an application to move through a dataset backward or forward by a specified number of records. This function takes only one parameter, the number of records by which to move. Positive integers indicate a forward move, while negative integers indicate a backward move.

For example, to move two records forward in Table1, use the following:

```
Table1.MoveBy(2);
```

When using this function, keep in mind that datasets are fluid entities, and the record which was five records back a moment ago may now be only four records back, or six records, or an unknown number of records, because multiple users may by simultaneously accessing the database and modifying its data.

Note There is no functional difference between calling *Table1.Next* and calling *Table1.MoveBy(1)*, just as there is no functional difference between calling *Table1.Prior* or calling *Table1.MoveBy(–1)*.

Modifying data in datasets

The following methods enable an application to insert, update, and delete data in datasets:

Table 3.4 Methods to insert, update and delete data in datasets

Method	Description
Edit	Puts the dataset into Edit state. If a dataset is already in Edit or Insert state, a call to *Edit* has no effect.
Append	Posts any pending data, moves current record to the end of the dataset, and puts the dataset in Insert state.
Insert	Posts any pending data, and puts the dataset in Insert state.
Post	Attempts to post the new or altered record to the database. If successful, the dataset is put in Browse state; if unsuccessful, the dataset remains in its current state.
Cancel	Cancels the current operation and puts the dataset into Browse state.
Delete	Deletes the current record and puts the dataset in Browse state.

The CanModify property

CanModify is a read-only property that specifies whether an application can modify the data in a dataset. When *CanModify* is False, then the dataset is read-only, and cannot be put into Edit or Insert state. When *CanModify* is True, the dataset can enter Edit or Insert state. Even if *CanModify* is True, it is not a guarantee that a user will be able to insert or update records in a table. Other factors may come in to play, for example, SQL access privileges.

TTable has a *ReadOnly* property that requests write privileges when set to False. When *ReadOnly* is True, *CanModify* will automatically be set to False. When *ReadOnly* is False, *CanModify* will be True if the database allows read and write privileges for the dataset and the underlying table. For more information, see “Using TTable.”

Posting data to the database

The *Post* method is central to a Delphi application’s interaction with a database table. *Post* behaves differently depending on a dataset’s state.

- In Edit state, *Post* modifies the current record.
- In Insert state, *Post* inserts or appends a new record.
- In SetKey state, *Post* returns the dataset to Browse state.

Posting can be done explicitly, or implicitly as part of another procedure. When an application moves off the current record, Delphi calls *Post* implicitly. Calls to the *First*, *Next*, *Prior*, and *Last* methods perform a *Post* if the table is in Edit or Insert state. The *Append* and *Insert* methods also implicitly perform a *Post* of any pending data.

Note *Post* is not called implicitly by the *Close* method. Use the *BeforeClose* event to post any pending edits explicitly.

Editing records

A dataset must be in Edit state before an application can modify records in the underlying table. The *Edit* method puts a dataset in Edit state. When in Edit state, the

Post method will change the current record. If a dataset is already in Edit state, a call to *Edit* has no effect.

The *Edit* and *Post* methods are often used together. For example,

```
Table1.Edit;
Table1.FieldByName('CustNo').AsString := '1234';
Table1.Post;
```

The first line of code in this example places the dataset in Edit mode. The next line of code assigns the string “1234” to the CustNo field. Finally, the last line posts, or writes to the database, the data just modified.

Adding new records

To add a new record to a dataset, an application can call either the *Insert* method or the *Append* method. Both methods put a dataset into Insert state. *Insert* opens a new, empty record after the current record. *Append* moves the current record to the end of the dataset and opens a new, empty record.

When an application calls *Post*, the new record will be inserted in the dataset in a position based on its index, if defined. Thus, for indexed tables, *Append* and *Insert* perform similarly. If no index is defined on the underlying table, then the record will maintain its position—so *Append* will add the record to the end of the table, and *Insert* will insert it at the cursor position when the method was called. In either case, posting a new record in a data grid may cause all the rows before and after the new record to change as the dataset follows the new row to its indexed position and then fetches data to fill the grid around it.

Deleting records

The *Delete* method deletes the current record from a dataset and leaves the dataset in Browse mode. The cursor moves to the following record.

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 Table is in Edit state, 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 Table’s *Cancel* method. A call to *Cancel* always returns a dataset to Browse state.

Working with entire records

The following methods enable an application to work with an entire record in one statement:

Table 3.5 Methods used to 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*.

Table 3.5 Methods used to work with entire records (continued)

Method	Description
InsertRecord([array of values])	Inserts the specified values as a record after 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 TFields. Application must perform a Post.

Each method takes a comma-delimited array of values as its argument, where each value corresponds to a column in the underlying table. The values can be literals, variables, null, or **nil**. 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 un-indexed tables, *AppendRecord* adds a record to the end of the table and *InsertRecord* inserts a record after the current cursor position. For indexed tables, both methods places 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. The application must first perform an *Edit* to put the dataset in Edit state. To modify the current record, it must then perform a *Post*.

Since these methods depend explicitly on the structure of the underlying tables, an application should use them only if the table structure will not change.

For example, the COUNTRY table has columns for Name, Capital, Continent, Area, and Population. If Table1 were linked to the COUNTRY table, the following statement would insert a record into the COUNTRY table:

```
Table1.InsertRecord(['Japan', 'Tokyo', 'Asia']);
```

The statement does not specify values for Area and Population, so it will insert Null values for these columns. 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:

```
Table1.Edit;
Table1.SetRecord(nil, nil, nil, 344567, 164700000);
Table1.Post;
```

This code assumes that the cursor will be positioned on the record just entered for Japan. It assigns values to the Area and Population fields and then posts them to the database. Notice the three **nil**s that act as place holders for the first three columns, which are not changed.

Setting the update mode

The *UpdateMode* property of a dataset determines how Delphi will find records being updated in a SQL database. This property is important in a multi-user environment when users may retrieve the same records and make conflicting changes to them.

When a user posts an update, Delphi uses the original values in the record to find the record in the database. This approach is similar to an optimistic locking scheme. *UpdateMode* specifies which columns Delphi uses to find the record. In SQL terms, *UpdateMode* specifies which columns are included in the WHERE clause of an UPDATE statement. If Delphi cannot find a record with the original values in the columns specified (if another user has changed the values in the database), Delphi will not make the update and will generate an exception.

The *UpdateMode* property may have the following values:

- *WhereAll* (the default): Delphi uses every column to find the record being updated. This is the most restrictive mode.
- *WhereKeyOnly*: Delphi uses only the key columns to find the record being updated. This is the least restrictive mode and should be used only if other users will not be changing the records being updated.
- *WhereChanged*: Delphi uses key columns and columns that have changed to find the record being updated.

For example, consider a COUNTRY table with columns for NAME (the key), CAPITAL, and CONTINENT. Suppose you and another user simultaneously retrieve a record with the following values:

- NAME = "Philippines"
- CAPITAL = "Nairobi"
- CONTINENT = "Africa"

Both you and the other user notice that the information in this record is incorrect and should be changed. Now, suppose the other user changes CONTINENT to "Asia," CAPITAL to "Manila," and posts the change to the database. A few seconds later, you change NAME to "Kenya" and post your change to the database.

If your application has *UpdateMode* set to *WhereKey* on the dataset, Delphi compares the original value of the key column (NAME = "Philippines") to the current value in the database. Since the other user did not change NAME, your update occurs. You think the record is now ["Kenya," "Nairobi," "Africa"] and the other users thinks it is ["Philippines," "Asia," "Manila"]. Unfortunately, it is actually ["Kenya," ," "Asia," "Manila"], which is still incorrect, even though both you and the other user think you have corrected the mistake. This problem occurred because you had *UpdateMode* set to its least restrictive level, which does not protect against such occurrences.

If your application had *UpdateMode* set to *WhereAll*, the Delphi would check all the columns when you attempt to make your update. Since the other user changed CAPITAL and CONTINENT, Delphi would not let you make the update. When you retrieved the record again, you would see the new values entered by the other user and realize that the mistake had already been corrected.

Bookmarking data

It is often useful to mark a particular location in a table so that you can quickly return to it when desired. Delphi provides this functionality through *bookmark* methods. These methods enable you to put a bookmark in the dataset, and quickly return to it later.

The three bookmarking methods are

- *GetBookmark*
- *GoToBookmark*
- *FreeBookmark*

These are used together. The *GetBookmark* function returns a variable of type *TBookmark*. A *TBookmark* contains a pointer to a particular location in a dataset. When given a bookmark, the *GoToBookmark* method will move an application's cursor to that location in the dataset.

FreeBookmark frees memory allocated for the specified bookmark. A call to *GetBookmark* allocates memory for the bookmark, so an application should call *FreeBookmark* before exiting, and before every use of a bookmark.

The following code illustrates a typical use of bookmarking:

```
procedure DoSomething;
var Bookmark: TBookmark;
begin
  Bookmark := Table1.GetBookmark; {allocate}
  Table1.DisableControls; {Disengage data controls}
  try
    Table1.First;
    while not Table1.EOF do
    begin
      {Do Something}
      Table1.Next;
    end;
  finally
    Table1.GotoBookmark(Bookmark);
    Table1.EnableControls;
    Table1.FreeBookmark(Bookmark); {deallocate}
  end;
end;
```

Notice the careful positioning of statements in this code. If the call to *GetBookmark* fails, controls are not disabled. If it succeeds, the bookmark is always freed and controls are always enabled.

Disabling, enabling, and refreshing data-aware controls

The *DisableControls* method disables all data-aware controls linked to a dataset. This method should be used with caution (for example, when programmatically iterating or searching through a dataset) to prevent "flickering" of the display as the cursor moves. As soon as the cursor is repositioned, an application should call the *EnableControls* method to re-enable data controls. It is important to re-enable controls with *EnableControls* as soon as the application completes its iteration or searching, to keep the form synchronized with the underlying dataset. Use a **try...finally** statement as in the example above.

The *Refresh* method flushes all local buffers and retrieves data from the specified dataset again. The dataset must be open. You can use this function to update a table if you think the table or the data it contains might have changed. Refreshing a table can sometimes lead to unexpected results. For example, if a user is viewing a record that has been deleted, then it will seem to disappear the moment the application calls *Refresh*. Similarly, data can appear to change while a user is viewing it if another user changes or deletes a record after the data was originally fetched and before a call to *Refresh*.

Using dataset events

Datasets have a number of events that enable an application to perform validation, compute totals, and perform other tasks depending on the method performed by the dataset. The events are listed in the following table.

Table 3.6 Dataset events

Event	Description
BeforeOpen, AfterOpen	Called before/after a dataset is opened.
BeforeClose, AfterClose	Called before/after a dataset is closed.
BeforeInsert, AfterInsert	Called before/after a dataset enters Insert state.
BeforeEdit, AfterEdit	Called before/after a dataset enters Edit state.
BeforePost, AfterPost	Called before/after changes to a table are posted.
BeforeCancel, AfterCancel	Called before/after the previous state is canceled.
BeforeDelete, AfterDelete	Called before/after a record is deleted.
OnNewRecord	Called when a new record is created; used to set default values.
OnCalcFields	Called when calculated fields are calculated.

For more information on these events and methods of the *TDataSet* component, refer to the online VCL reference.

Abort a method

To abort a method such as an *Open* or *Insert*, raise an exception or call the *Abort* procedure in any of the Before methods (*BeforeOpen*, *BeforeInsert*, and so on). For example, the following code confirms a delete operation:

```
procedure TForm1.TableBeforeDelete (Dataset: TDataset);
begin
   if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel, 0) = mrYes then
      Abort;
end;
```

Using OnCalcFields

The *OnCalcFields* event is used to set the values of calculated fields. The *AutoCalcFields* property determines when *OnCalcFields* is called. If *AutoCalcFields* is True, then *OnCalcFields* is called when:

- The dataset is opened.
- Focus moves from one visual component to another, or from one column to another in a *DBDataGrid*.

- A record is retrieved from the database.

OnCalcFields is also called whenever a non-calculated field's value changes, regardless of the setting of *AutoCalcFields*.

Typically, the *OnCalcFields* event will be called often, so it should be kept short. Also, if *AutoCalcFields* is True, *OnCalcFields* should not perform any actions that modify the dataset (or the linked dataset if it is part of a master-detail relationship), because this can lead to recursion. For example, if *OnCalcFields* performs a *Post*, and *AutoCalcFields* is True, then *OnCalcFields* will be called again, leading to another *Post*, and so on.

If *AutoCalcFields* is False, then *OnCalcFields* is called when the dataset's *Post* method is called (or any method that implicitly calls *Post*, such as *Append* or *Insert*).

While the *OnCalcFields* event is executed, a dataset will be put in *CalcFields* mode. When a dataset is in *CalcFields* mode, you cannot set the values of any fields other than calculated fields. After *OnCalcFields* is completed, the dataset will return to its previous mode.

Using TTable

TTable is one of the most important database component classes. Along with the other dataset component class, *TQuery*, it enables an application to access a database table. This section describes the most important properties that are unique to *TTable*.

Specifying the database table

TableName specifies the name of the database table to which the *TTable* component is linked. You can set this property at design time through the Object Inspector.

The *DatabaseName* property specifies where Delphi will look for the specified database table. It can be a BDE alias, an explicit specification, or the *DatabaseName* defined by any *TDatabase* component in the application. For Paradox and dBASE tables, an explicit specification is a directory path; for SQL tables, it is a directory path and database name.

Instead of an actual directory path or database name, *DatabaseName* can also be a BDE alias. The advantage of this is that you can change the data source for an entire application by simply changing the alias definition in the BDE Configuration Utility. For more information on using the BDE Configuration Utility, see Appendix B, "Using the BDE configuration utility." For more information on the *DatabaseName* property, see the online VCL reference.

Note Neither of these properties can be changed when a table is open—that is, when the table's *Active* property is set to a value of True.

The TableType property

The *TableType* property specifies the type of the underlying database table. This property is not used for SQL tables.

If *TableType* is set to *Default*, the table's file-name extension determines the table type:

- Extension of .DB or no file-name extension: Paradox table
- Extension of .DBF : dBASE table
- Extension of .TXT : ASCII table

If the value of *TableType* is not *Default*, then the table will always be of the specified *TableType*, regardless of file-name extension.

Searching a table

TTable has a number of functions that will search for values in a database table:

- Goto functions
- Find functions

The easiest way to search for values is with the Find functions, *FindKey* and *FindNearest*. These two functions combine the functionality of the basic Goto functions, *SetKey*, *GoToKey*, and *GoToNearest*, which are described first.

In dBASE and Paradox tables, these functions can search only on index fields. In SQL tables, they can search on any fields, if the field name is specified in the *IndexFieldNames* property of the *TTable*. For more information, see "Indexes" on page 74.

To search a dBASE or Paradox table for a value in a non-index field, use SQL SELECT syntax with a *TQuery* component. For more information on using SQL and *TQuery* components, see Chapter 5, "Using SQL in applications."

Using Goto functions

The *GoToKey* and *GoToNearest* methods enable an application to search a database table using a key. *SetKey* puts a table in "search mode," more accurately referred to as SetKey state. In SetKey state, assignments indicate values for which to search for in indexed fields. *GoToKey* then moves the cursor to the first row in the table that matches those field values.

For example, the following code could be used in the *OnClick* event of a button:

```
procedure TSearchDemo.SearchExactClick(Sender: TObject);
begin
Table1.SetKey; {First field is the key}
Table1.Fields[0].AsString := Edit1.Text;
Table1.GoToKey;
end;
```

The first line of code after **begin** puts *Table1* in SetKey state. This indicates that the following assignment to the table's *Fields* property specifies a search value. The first column in the table, corresponding to *Fields*[0], is the index. In this example, the value the application searches for is determined by the text the user types into the edit control, *Edit1*. Finally, *GoToKey* performs the search, moving the cursor to the record if it exists.

GoToKey is a Boolean function that moves the cursor and returns True if the search is successful. If the search is unsuccessful, it returns False and does not change the position of the cursor. For example,

```
Table1.SetKey;
Table1.Fields[0].AsString := 'Smith';
if not Table1.GotoKey
   then ShowMessage('Record Not Found');
```

If the search does not find a record with a first column matching "Smith," the *ShowMessage* function displays a dialog box with the "Record Not Found" message.

If a table has more than one key column, and you want to search for values in a sub-set of the keys, set *KeyFieldCount* to the number of columns on which you are searching. For example, if a table has three columns in its primary key, and you want to search for values in just the first, set *KeyFieldCount* to 1. For tables with multiple-column keys, you can search only for values in contiguous columns, beginning with the first. That is, you can search for values in the first column, or the first and second, or the first, second, and third, but not just the first and third.

GoToNearest is similar, except it finds the nearest match to a partial field value. It can be used only for columns of string data type. For example,

```
Table1.SetKey;
Table1.Fields[0].AsString := 'Sm';
Table1.GoToNearest;
```

If a record exists with "Sm" as the first two characters, the cursor will be positioned on that record. Otherwise, the position of the cursor does not change and *GoToNearest* returns False.

If it is not searching on the primary index of a local table, then an application must specify the column names to use in the *IndexFieldNames* property or the name of the index to use in the *IndexName* property of the table. For example, if the CUSTOMER table had a secondary index named "CityIndex" on which you wanted to search for a value, you would need to set the value of the table's *IndexName* property to "CityIndex." You could then use the following syntax when you searching on this field:

```
Table1.IndexName := 'CityIndex';
Table1.Open;
Table1.SetKey;
Table1.FieldByName('City').AsString := Edit1.Text;
Table1.GoToNearest;
```

Because indexes often have non-intuitive names, you can use the *IndexFieldNames* property instead to specify the names of indexed fields.

Each time an application calls *SetKey*, it must set all the field values for which it will search. That is, *SetKey* clears any existing values from previous searches. To keep previous values, use *EditKey*.

For example, to extend the above search to find a record with the specified city name in a specified country, an application could use the following code:

```
Table1.EditKey;
Table1.FieldByName('Country').AsString := Edit2.Text;
```

Using Find functions

The Find functions, *FindKey* and *FindNearest* provide easy way to search a table. They combine the functionality of *SetKey*, field assignment, and Goto functions into a single statement.

Each of these methods takes a comma-delimited array of values as its argument, where each value corresponds to a column in the underlying table. The values can be literals, variables, null, or **nil**. If the number of values in an argument is less than the number of columns in the database table, then the remaining values are assumed to be null. The functions will search for values specified in the array in the current index.

FindKey is similar to *GotoKey*:

- It will put a table in search mode (SetKey state).
- It will find the record in the table that matches the specified values. If a matching record is found, it moves the cursor there, and returns True.
- If a matching record is not found, it does not move the cursor, and returns False.

For example, if Table1 is indexed on its first column, then the statement:

```
Table1.FindKey([Edit1.Text]);
```

will perform the same function as the three statements:

```
Table1.SetKey; {First field is the key}
Table1.Fields[0].AsString := Edit1.Text;
Table1.GoToKey;
```

FindNearest is similar to *GotoNearest*, in that it will move the cursor to the row with the nearest matching value. This can be used for columns of string data type only.

Both of these functions work by default on the primary index column. To search the table for values in other indexes, you must specify the field name in the table's *IndexFieldNames* property or the name of the index in the *IndexName* property.

Note With Paradox or dBASE tables, these methods work only with indexed fields. With SQL databases, they can work with any columns specified in the *IndexFieldNames* property.

The KeyExclusive property

The *KeyExclusive* property indicates whether a search will position the cursor on or after the specified record being searched for. If *KeyExclusive* is False (the default), then *GoToNearest* and *FindNearest* will move the cursor to the record that matches the specified values. If True, then the search functions will go the record immediately following the specified key, if found.

Limiting records retrieved by an application

Tables in the real world can be huge, so applications often need to limit the number of rows they work with. The following methods of *TTable* enable an application to work with a subset of the data in a database table:

- *SetRangeStart* and *EditRangeStart*
- *SetRangeEnd* and *EditRangeEnd*
- *SetRange*([*Start Values*], [*End Values*])
- *ApplyRange*
- *CancelRange*

SetRangeStart indicates that subsequent assignments to field values will specify the start of the range of rows to include in the application. *SetRangeEnd* indicates that subsequent assignments will specify the end of the range of rows to include. Any column values not specified are not considered. The corresponding methods *EditRangeStart* and *EditRangeEnd* indicate to keep existing range values and update with the succeeding assignments.

ApplyRange applies the specified range. If *SetRangeStart* has not been called when *ApplyRange* is called, then the start range will be the beginning of the table; likewise, if *SetRangeEnd* has not been called, the end range will be the end of the table. *CancelRange* cancels the range filter and includes all rows in the table.

The *SetRange* function combines *SetRangeStart*, *SetRangeEnd*, and field assignments into a single statement that takes an array of values as its argument.

Note With Paradox or dBASE tables, these methods work only with indexed fields. With SQL databases, they can work with any columns specified in the *IndexFieldNames* property.

For example, suppose there is a form with a *TTable* component named Cust, linked to the CUSTOMER table. CUSTOMER is indexed on its first column (CustNo). The form also has two Edit components named *StartVal* and *EndVal*, and you have used the Fields Editor to create a *TField* component for the CustNo column. Then these methods could be applied (for example, in a button's *OnClick* event) as follows:

```
Cust.SetRangeStart;
CustCustNo.AsString := StartVal.Text;
Cust.SetRangeEnd;
if EndVal.Text <> '' then
    CustCustNo.AsString := EndVal.Text;
Cust.ApplyRange;
```

Notice that this code first 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 table will be included, since all values are greater than null. However, if the text entered for *EndVal* is null, then no records will be included, since none are less than null.

This code could be re-written using the *SetRange* function as follows:

```
if EndVal.Text <> '' then
   Cust.SetRange([StartVal.Text], [EndVal.Text]);
Cust.ApplyRange;
```

Using partial keys

If a key is composed of one or more string fields, these methods support partial keys. For example, if an index is based on the LastName and FirstName columns, the following range specifications are valid:

```
Table1.SetRangeStart;
Table1.FieldByName('LastName').AsString := 'Smith';
Table1.SetRangeEnd;
Table1.ApplyRange;
```

This will include all records where LastName greater than or equal to "Smith." The value specification could also be:

```
Table1.FieldByName('LastName').AsString := 'Sm';
```

This would include records which have LastName greater than or equal to"Sm." The following would include records with a LastName greater than or equal to "Smith" and a FirstName greater than or equal to "J":

```
Table1.FieldByName('LastName').AsString := 'Smith';
Table1.FieldByName('FirstName').AsString := 'J';
```

The KeyExclusive property

The *KeyExclusive* property determines whether the filtered range excludes the range boundaries. The default is False, which means rows will be in the filtered range if they are greater than or equal to the start range specified and less than or equal to the end range specified. If *KeyExclusive* is True, the methods will filter strictly greater than and less than the specified values.

Indexes

An *index* determines how records are sorted when a Delphi application displays data. By default, Delphi displays data in ascending order, based on the values of the primary index column(s) of a table.

Delphi supports SQL indexes, maintained indexes for Paradox tables, and maintained .MDX (production) indexes for dBASE tables. Delphi does not support:

- Non-maintained indexes on Paradox tables.
- Non-maintained or .NDX indexes of dBASE tables.
- The *IndexFieldCount* property for a dBASE table opened on an expression index.

The *GetIndexNames* method returns a list of the names of available indexes on the underlying database table. For Paradox tables, the primary index is unnamed and therefore not returned by *GetIndexNames*. To use a primary index on a Paradox table, set the corresponding *TTable*'s *IndexName* to a null string.

IndexFields is an array of field names used in the index. *IndexFieldCount* is the number of fields in the index. *IndexFieldCount* and *IndexFields* are read-only properties that are available only during run-time.

Use the *IndexName* property to sort or search a table on an index other than the primary index. In other words, to use the primary index of a table, you need do nothing with the *IndexName* property. To use a secondary index, however, you must specify it in *IndexName*.

For tables in a SQL database, the *IndexFieldNames* property specifies the columns to use in the ORDER BY clause when retrieving data. The entry for this property is a semicolon-delimited list of field names. Records are sorted by the values in the specified fields. Sorting can be only in ascending order. Case-sensitivity depends on the server being used.

For example, to sort customer records in an SQL table by zip code and then by customer number, enter the following for the *IndexFieldNames* property:

```
ZipCode;CustNo
```

For Paradox and dBASE tables, Delphi will pick an index based on the columns specified in *IndexFieldNames*. An error will occur if you specify a column or columns that cannot be mapped to an existing index.

The *IndexName* and *IndexFieldNames* properties are mutually exclusive. Setting one property clears the value of the other.

The Exclusive property

The *Exclusive* property indicates whether to open the table with an exclusive lock. If True, no other user will be able to access it at the same time. You cannot open a table in Exclusive mode if another user is currently accessing the table.

If the underlying table is in a SQL database, an exclusive table-level lock may allow others to read data from the table but not modify it. Some servers may not support exclusive table-level locks, depending on the server. Refer to your server documentation for more information.

Other properties and methods

In addition to dataset properties shared with *TQuery*, *TTable* has a number of unique methods and properties. For example, the unique methods include

- *EmptyTable*, which deletes all records (rows) in the table.
- *DeleteTable*, which deletes the table.
- *BatchMove*, which copies data and table structures from one table to another, similar to the operation of *TBatchMove*.

A few of the more important properties and methods are discussed in this section. For a complete list and descriptions, see the online *VCL Reference*.

The ReadOnly and CanModify properties

Before opening a *TTable*, set *ReadOnly* False to request read and write privileges for the dataset. Set *ReadOnly* to True to request read-only privileges for the dataset. Depending on the characteristics of the underlying table, the request for read and write privileges may or may not be granted by the database.

CanModify is a read-only property of datasets that reflects the actual rights granted for the dataset. When *ReadOnly* is True, *CanModify* will automatically be set to False. When *ReadOnly* is False, *CanModify* will be True if the database allows read and write privileges for the dataset and the underlying table.

When *CanModify* is False, then the table is read-only, and the dataset cannot be put into Edit or Insert state. When *CanModify* is True, the dataset can enter Edit or Insert state. Even if *CanModify* is True, it is not a guarantee that a user will be able to insert or update records in a table. Other factors may come in to play, for example, SQL access privileges.

The GoToCurrent method

GoToCurrent is a method that synchronizes two *TTable* components linked to the same database table and using the same index. This method takes a *TTable* component as its argument, and sets the cursor position of the *TTable* to the current cursor position of the argument. For example,

```
Table1.GotoCurrent(Table2);
```

Creating master-detail forms

The *MasterSource* and *MasterFields* are used to define one-to-many relationships between two tables. The *MasterSource* property is used to specify a data source from which the table will get data for the master table. For instance, if you link two tables in a master-detail relationship, then the detail table can track the events occurring in the master table by specifying the master table's *TDataSource* in this property.

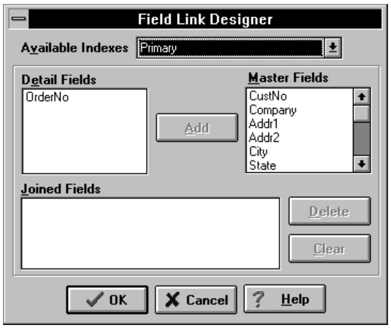
To link tables based on values in multiple column names, use a semicolon delimited list:

```
Table1.MasterFields := 'OrderNo;ItemNo';
```

The Field Link Designer

At design time, when you double-click (or click on the ellipsis button) on the *MasterFields* property in the Object Inspector, the Field Link Designer dialog box opens.

Figure 3.3 Field Link designer



The Field Link Designer provides a visual way to link master and detail tables. The Available Indexes combo box shows the currently selected index by which to join the two tables. For Paradox tables, this will be "Primary" by default, indicating that the primary index of the detail field will be used. Any other named indices defined on the table will be shown in the drop-down list.

Select the field you want to use to link the detail table in the Detail Fields list, the field to link the master table in the Master Fields list, and then choose Add. The selected fields will be displayed in the Joined Fields list box. For example,

```
OrderNo -> OrderNo
```

For tables on a database server, the Available Indexes combo box will not appear, and you must select the detail and master fields to join manually in the Detail Fields and Master Fields list boxes.

Using TDataSource

TDataSource acts as a conduit between datasets and data-aware controls. Often the only thing you will do with a *TDataSource* component is to set its *DataSet* property to an appropriate dataset object. Then you will set data controls' *DataSource* property to the specific *TDataSource*. You also use *TDataSource* components to link datasets to reflect master-detail relationships.

Using TDataSource properties

TDataSource has only a few published properties in addition to the standard *Name* and *Tag* properties.

The DataSet property

The *DataSet* property specifies the name of the dataset from which the *TDataSource* will get its data. You can also set the *DataSet* property to a dataset on another form to synchronize the data controls on the two forms. For example,

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

The Enabled property

The *Enabled* property can temporarily disconnect a *TDataSource* from its *TDataSet*. When set to False, all data controls attached to the data source will go blank and become inactive until *Enabled* is set to True.

In general, it is recommended to use datasets' *DisableControls* and *EnableControls* methods to perform this function, because they affect all attached data sources.

The AutoEdit property

The *AutoEdit* property of *TDataSource* specifies whether datasets connected to the data source automatically enter Edit state when the user starts typing in data-aware controls linked to the dataset. If *AutoEdit* is True (the default), Delphi automatically puts the dataset in Edit state when a user types in a linked data-aware control. Otherwise, a dataset enters Edit state only when the application explicitly calls its *Edit* method. For more information on dataset states, see "Dataset states" on page 58.

Using TDataSource events

TDataSource has three events associated with it:

- *OnDataChange*
- *OnStateChange*
- *OnUpdateData*

The OnDataChange event

OnDataChange is called whenever the cursor moves to a new record. In other words, if an application calls *Next*, *Previous*, *Insert*, or any method that leads to a change in the cursor position, then an *OnDataChange* is triggered.

This event is useful if an application is keeping components synchronized manually.

The OnUpdateData event

OnUpdateData is called whenever the data in the current record is about to be updated. For instance, an *OnUpdateData* event will occur after *Post* is called but before the data is actually posted to the database.

This event is useful if an application uses a standard (non-data aware) control and needs to keep it synchronized with a dataset.

The OnStateChange event

OnStateChange is called whenever the mode (state) of a data source's dataset changes. A dataset's *State* property records its current state. This event is useful for performing actions as a *TDataSource*'s state changes, as the following examples illustrate.

During the course of a normal database session, a dataset's state will change frequently. To track these changes, you can use code in *OnStateChange* such as the following example that displays the current state in a Label component:

```
procedure TForm1.DataSource1.StateChange(Sender:TObject);
var
S:String;
begin
case Table1.State of
dsInactive: S := 'Inactive';
dsBrowse: S := 'Browse';
dsEdit: S := 'Edit';
dsInsert: S := 'Insert';
dsSetKey: S := 'SetKey';
end;
Label1.Caption := S;
end;
```

Similarly, *OnStateChange* can be used to enable or disable buttons or menu items based on the current state. For example,

```
procedure Form1.DataSource1.StateChange(Sender: TObject);
begin
   InsertBtn.Enabled := (Table1.State = dsBrowse);
```

```
        CancelBtn.Enabled := Table1.State in [dsInsert, dsEdit, dsSetKey];
        ...
    end;
```

Using TFields and the Fields Editor

TField components correspond to database columns. They are created

- At run time by Delphi whenever a dataset component is active. This creates a dynamic set of *TFields* that mirrors the columns in the table at that time.
- At design time through the Fields Editor. This creates a persistent set of *TFields* that does not change, even if the structure of the underlying table changes.

There are *TField* components corresponding to all possible data types, including *TStringField, TSmallintField, TIntegerField, TWordField, TBooleanField, TFloatField, TCurrencyField, TBCDField, TDateField, TTimeField,* and *TDateTimeField*. This chapter discusses *TFields* in general, and the discussion applies to all the different sub-types. For information on the properties of a specific type, see the online *VCL Reference*.

What are TField components?

All Delphi data-aware components rely on an underlying object class, *TField*. Although not visible on forms, *TField* components are important because they provide an application a direct link to a database column. *TFields* contain properties specifying a column's data type, current value, display format, edit format, and other characteristics. *TField* components also provide events, such as *OnValidate*, that can be used to implement field-based validation rules.

Each column retrieved from a table has a corresponding *TField* component. By default, *TField* components are dynamically generated at design time when the *Active* property of a *TTable* or *TQuery* component is set to True. At run time, these components are also dynamically generated. Dynamic generation means Delphi builds *TField* components based on the underlying physical structure of a database table each time the connection to the table is activated. Thus, dynamically generated *TFields* always correspond to the columns in the underlying database tables.

To generate a persistent list of *TField* components for an application, use the Fields Editor. Using the Fields Editor to specify a persistent list of *TField* components is smart programming. Creating *TField* components with the Fields Editor provides efficient, readable, and type-safe programmatic access to underlying data. It guarantees that each time your application runs, it uses and displays the same columns, in the same order, every time, even if the physical structure of the underlying database has changed. Creating *TField* components at design time guarantees that data-aware components and program code that rely on specific fields always work as expected. If a column on which a persistent *TField* component is based is deleted or changed, then Delphi generates an exception rather than running the application against a non-existent column or mismatched data.

Using the Fields Editor

When *TTable* and *TQuery* components are connected to a database and their *Active* properties are set to True, they dynamically generate a *TField* component for each column in a table or query. Each *TField* component stores display-related information about a field. The display information is used by data control components, such as *TDBEdit* and *TDBGrid*, to format data for display in a form. You can make *TField* components persistent and edit their display characteristics by invoking the Fields Editor.

The Fields Editor enables you to:

- Generate a persistent list of *TField* components.
- Modify the display properties of persistent *TField* components.
- Remove *TField* components from the list of persistent components.
- Add new *TField* components based on existing columns in a table.
- Define calculated *TField* components that behave just like physical data columns, except that their values are computed programmatically.

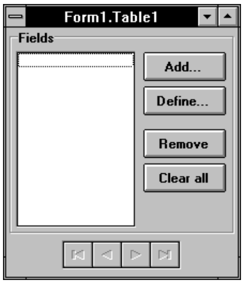
Starting the Fields Editor

To invoke the Fields Editor for a *TTable* or *TQuery* component,

- Double-click the mouse on the component, or
- Select the component, right click the mouse, and select Fields Editor from the pop-up menu.

The Fields Editor opens, with the name of the Form and Table on which it was invoked in the title bar:

Figure 3.4 Fields Editor



The Fields list box displays the names of persistent *TField* components associated with the data access component. The first time you invoke the Fields Editor on a particular a *TTable* or *TQuery* component, the Fields list is empty because all *TFields* are dynamically created. If any *TField* objects are listed in *Fields*, then data-aware components can only display data from those fields. You can drag and drop individual *TField* objects within the *Field* list box to change the order in which fields are displayed in controls, like *TDBGrid*, that display multiple columns.

The navigator buttons at the bottom of the Fields Editor window enable you to scroll through the records one at a time, and to jump to the first or last record of the dataset if it is active.

The Add button enables you to see a list of column names in the physical dataset but not already included in the Fields list, and to create new *TField* components for them.

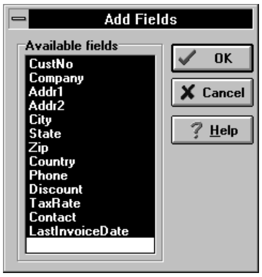
The Define button enables you to create calculated fields. Fields created this way are only for display purposes. The underlying physical structure of the table or data is not changed.

The Remove button deletes the selected *TFields*. The Clear All button deletes all the *TFields* shown in the Fields list.

Adding a TField component

The Add button of the Fields Editor enables you to specify which *TField* components will be included in a dataset. To see a list of fields currently available to a *TTable* or *TQuery* component, click the Add button. The Add Fields dialog box appears.

Figure 3.5 Fields Editor Add Fields dialog box



The *Available Fields* list box shows all database fields that do not have persistent *TFields* instantiated. Initially, all available fields are selected. Use the mouse to select specific fields and then choose OK. The selected fields move to the *Fields* list box in the main Fields Editor window.

Fields moved to the *Available Fields* list become persistent. Each time the dataset is opened, Delphi verifies that each non-calculated field exists or can be created from data in the database. If it cannot, an exception is raised, warning you that the field is not valid, and the dataset is not opened.

Deleting a TField component

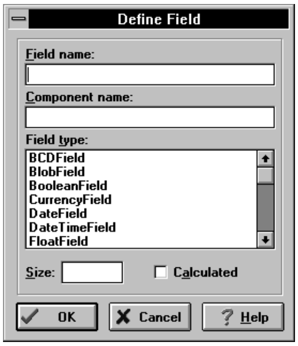
The Remove button of the Fields Editor deletes the selected *TField* components from the *Fields* list box. Fields removed from the *Field* list box are no longer in the dataset and cannot be displayed by data-aware components. Removing a *TField* component is useful to display a subset of available fields within a table, or when you want to define your own field to replace an existing field.

You can re-create deleted *TField* components with the Add button, but any changes previously made to its properties or events will be lost.

Defining a new TField component

The Define button of the Fields Editor enables you to create new *TField* components for display. You can create a new *TField* based on a column in the underlying table (for example, to change the data type of the field), but its main purpose is to create new *TField* components whose values are calculated programmatically.

Figure 3.6 Define Field dialog box



Defining a calculated field

A calculated field is used to display values calculated at run time in the dataset's *OnCalcFields* event handler. For example, you might create a string field that displays concatenated values from two other fields.

To create a calculated field:

1. Choose the Define button in the Fields Editor window.
2. Enter the name of the new field in the Field Name edit box, or select a field name from the drop-down list. A corresponding *TField* component name appears automatically in the Component edit box as you type. This name is the identifier you use to access the field programmatically.
3. Select the data type for the field from the Field Type list box.
4. Check the Calculated check box if it is not already checked.
5. Choose OK. The newly defined calculated field is automatically added to the Fields list box in the main Fields Editor window, and the component declaration is automatically added to the form's **type** declaration in the source code.

To edit the properties or events associated with the new *TField* component, select the component name in the Fields list box, then edit the properties or events via the Object Inspector.

Programming a calculated field

Once a calculated field is defined, it has a null value unless you write code to provide values for the field. Code for a calculated field is placed in the *OnCalcFields* event for its dataset. An *OnCalcFields* event handler can only modify fields that have a *Calculated* property of True.

For example, on Form1, the *OnCalcFields* event for a *TTable* component named Table1 is

```
TForm1.Table1CalcFields
```

To program a value for a calculated field:

1 Select the *TTable* or *TQuery* component from the Object Inspector drop-down list.

2 Choose the Object Inspector Events tab.

3 Double-click the *OnCalcFields* property to bring up or create a *CalcFields* procedure for the *TTable* or *TQuery* component.

4 Write the code that sets the values and other properties of the calculated field as desired.

Editing a TField component

You can customize properties and events of persistent *TField* components at design time. To do this, select the *TField* either in the Fields list box of the Fields Editor or the component list of the Object Inspector. 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.

Editing Display properties

To edit the display properties of a selected *TField* component, click the Properties tab on the Object Inspector window. The following table summarizes display properties that can be edited. Not all properties appear for all *TField* descendents.

Table 3.7 TField properties

Property	**Purpose**
Alignment	Displays contents of field left justified, right justified, or centered within a data-aware component.
Calculated	True, field value can be calculated by a *CalcFields* method at run time. False, field value is determined from the current record.
Currency	True, numeric field displays monetary values. False, numeric field 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 *TDBGrid*.
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, etc.).
FieldName	Specifies the actual name of column in the physical table from which the *TField* component derives its value and data type.
Index	Specifies the order of the field in a dataset.
MaxValue	Specifies the maximum numeric value that can be entered in an editable numeric field.

Table 3.7 TField properties (continued)

Property	Purpose
MinValue	Specifies the minimum numeric value that can be entered in an editable numeric field.
Name	Specifies the component name of the *TField* component within Delphi.
ReadOnly	True: Field can be displayed in a component, but cannot be edited by a user. False: Field can be displayed and edited.
Size	Specifies the maximum number of characters that can be displayed or entered in a string-based field, or the size of byte and var byte fields.
Tag	Long integer bucket available for programmer use in every component as needed.
Visible	True: Field is displayed by a *TDBGrid* component. User-defined components can also make display decisions based on this property. False: Field is not displayed by a *TDBGrid* component.

Not all properties are available to all *TField* components. For example, a component of type *TStringField* does not have *Currency*, *MaxValue*, or *DisplayFormat* properties. A component of type *TFloatField* does not have a *Size* property.

Boolean properties, those that can be toggled between True and False, can be changed by double-clicking the property in the Object Inspector. Other properties require you to enter values or pick from drop-down lists in the Object Inspector. All *TField* properties can also be manipulated programmatically.

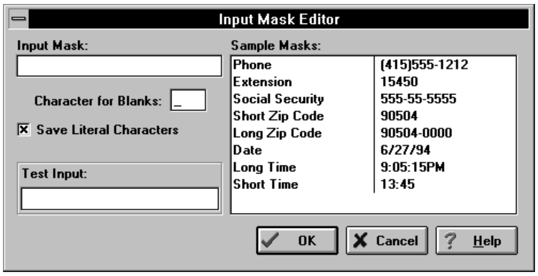
While the purpose of most properties is straight-forward, some properties, such as *Calculated*, require additional programming steps to be useful. Others, such as *DisplayFormat*, *EditFormat*, and *EditMask*, are interrelated; their settings must be coordinated. For more information about using the *Calculated* property, see "Programming a calculated field." For more information about using *DisplayFormat*, *EditFormat*, and *EditMask*, see "Formatting fields."

Using the Input Mask Editor

The *EditMask* property provides a way to limit the entries that a user can type into data aware controls tied to a *TField*. You can enter a specific edit mask by hand or use the Input Mask Editor to create a mask.

To use the Input Mask Editor, when you have selected a *TField* component, double-click on the EditMask field in the Object Inspector or click the ellipsis button in the Values column. The Input Mask Editor opens:

Figure 3.7 Input Mask Editor



Phone	(415)555-1212
Extension	15450
Social Security	555-55-5555
Short Zip Code	90504
Long Zip Code	90504-0000
Date	6/27/94
Long Time	9:05:15PM
Short Time	13:45

To use one of the sample masks, select it in the Sample Masks list box. You can then customize the mask as desired by editing it in the Input Mask text field. You can test the allowable input in the Test Input field. For more information, see the online Help.

Note For the *TStringField*, the *EditMask* property is also used as a display format.

Formatting fields

Delphi provides built-in display and edit format routines and intelligent default formatting for *TField* components. These routines and formats require no action on the programmer's part. Default formatting conventions are based on settings in the Windows Control Panel. For example, using default Windows settings in 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. Only format properties appropriate to the data type of a *TField* component are available for a given component.

All *TField* component formatting is performed by the following routines:

Table 3.8 TField formatting routines

Routine	Used by . . .
FormatFloat	*TFloatField*, *TCurrencyField*
FormatDateTime	*TDateField*, *TTimeField*, *TDateTimeField*
FormatInteger	*TIntegerField*, *TSmallIntField*, *TWordField*

The format routines use the International settings specified in the Windows Control Panel for determining how to display date, time, currency, and numeric values. You can edit the *DisplayFormat* and *EditFormat* properties of a *TField* component to override the default display settings for a *TField*, or you can handle the *OnGetText* and *OnSetText* events for a *TField* to do custom programmatic formatting.

Handling TField events

To edit the events for a selected *TField* component, click the Events tab on the Object Inspector window. The following table summarizes events that can be edited:

Table 3.9 Published TField events

Event	Purpose
OnChange	Called when the value for a TField component changes.
OnGetText	Called when the value for a TField component is retrieved for display or editing.
OnSetText	Called when the value for a TField component is set.
OnValidate	Called to validate the value for a TField 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 Delphi's built-in formatting functions,*FormatFloat*, *FormatDate*, and so on.

TFields have a *'FocusControl* method that enables an event to set focus to the first data-aware control associated with the *TField*. This is especially important for record-oriented validation (for example, on *BeforePost* of a *TTable*) since a *TField* may be validated whether or not its associated data control has focus.

Using TField conversion functions

TFields have built-in functions for conversion among data types. Depending on the *TField* type, different conversion functions are available and do different things. The following table summarizes these functions.

Table 3.10 TField conversion functions

TField Type	AsString	AsInteger	AsFloat	AsDateTime	AsBoolean
TStringField	String type by definition	Convert to Integer if possible	Convert to Float if possible	Convert to Date if possible	Convert to Boolean if possible
TIntegerField **TSmallIntField** **TWordField**	Convert to String	Integer type by definition	Convert to Float	Not Allowed	Not Allowed
TFloatField **TCurrencyField** **TBCDField**	Convert to String	Round to nearest integer value	Float type by definition	Not Allowed	Not Allowed
TDateTimeField **TDateField** **TTimeField**	Convert to String. Content depends on DisplayFormat of Field	Not Allowed	Convert Date to number of days since 01/01/0001 Convert Time to fraction of 24 hours	DateTime type by definition Zero date or time if not specified	Not Allowed
TBooleanField	Convert to String "True" or "False"	Not Allowed	Not Allowed	Not Allowed	Boolean type by definition
TBytesField **TVarBytesField** **TBlobField** **TMemoField** **TGraphicField**	Convert to String (Generally only makes sense for TMemoField)	Not Allowed	Not Allowed	Not Allowed	Not Allowed

The conversion functions can be used in any expression involving a *TField* component, on either side of an assignment statement. For example, the following statement converts the value of the *TField* named *MyTableMyField* to a string and assigns it to the text of the *Edit1* control:

```
Edit1.Text := MyTableMyField.AsString;
```

Conversely, this statement assigns the text of the *Edit1* control to the *TField* as a string:

```
MyTableMyField.AsString := Edit1.Text;
```

An exception occurs if an unsupported conversion is performed at run time.

Accessing TField properties programmatically

An application can access the value of a database column through a *TField* component's *Value* property. For example, the following statement assigns the value of the CustTableCountry *TField* to the text in the *TEdit* component Edit3:

```
Edit3.Text :=  CustTableCountry.Value;
```

Any properties of *TField* components that are available from the Object Inspector can also be accessed and adjusted programmatically as well. For example, this statement changes field ordering by setting the *Index* property of CustTableCountry to 3:

```
CustTableCountry.Index := 3;
```

Displaying data with standard controls

You can display database values at run time with standard components as well as data aware and *TField* components. Besides accessing *TField* components created with the Fields Editor, there are two ways to access column values at run time: the *Fields* property and the *FieldsByName* method. Each accesses the value of the current row of the specified column in the underlying database table at run time. Each requires a dataset component in the form, but not a *TDataSource* component.

In general, you should use the data-aware controls built in to Delphi in database applications. These components have properties and methods built in to them that enable them to be connected to database columns, display the current values in the columns, and make updates to the columns. If you use standard components, you must provide analogous code by hand.

Using the Fields property

You can access the value of a field with the *Fields* property of a dataset component, using as a parameter the ordinal number of the column in the table (starting at 0). To access or change the field's value, convert the result with the appropriate conversion function, such as *AsString* or *AsInteger*.

This method requires you to know the order and data types of the columns in the table. Use this method if you want to iterate over a number of columns or if your application works with tables that are not available at design time.

For example, the following statement assigns the current value of the seventh column (Country) in the *CustTable* table to the *Edit1* component:

```
Edit1.Text := CustTable.Fields[6].AsString;
```

Conversely, you can assign a value to a column a dataset in Edit mode by assigning the appropriate *Fields* property to the value of a component. For example,

```
CustTable.Fields[6].AsString := Edit1.Text;
```

To make these assignments occur, you must enter them in an event such as a *TButton OnClick* event, or an edit control's *OnEnter* event.

Using the FieldByName method

You can access the value of a field with the *FieldByName* method by specifying the dataset component name, and then passing *FieldByName* the name of the column you want to access. To access or change the field's value, convert the result with the appropriate conversion function, such as *AsString* or *AsInteger*.

This method requires you to know the name of the column you want to access or if your application works with tables that are not available at design time.

For example, the following statement assigns the value of the Country column in the CustTable table to Edit2:

```
Edit2.Text := CustTable.FieldByName('Country').AsString;
```

Conversely, you can assign a value to a column a dataset in Edit mode by assigning the appropriate *FieldByName* property to the value of a component. For example,

```
CustTable.FieldByName('Country').AsString := Edit2.Text;
```

To make these assignments occur, you must enter them in an event such as a button's *OnClick* or a Edit component's *OnExit* event.

Incorporating reports in an application

Delphi applications can include reports created with ReportSmith with the *TReport* component. *TReport* appears on the Data Access component page. To incorporate a report in an application, simply add a *TReport* component to the desired form as you would any other component. Then specify the name of the report (created with ReportSmith) and other report parameters with properties of the component.

Designing reports with ReportSmith is described in *Creating Reports*. You can invoke ReportSmith at design time by double-clicking on a *TReport* component, or on the ReportSmith icon in the Delphi program group.

Specify the name of an existing report in the *ReportName* property and the directory in the *ReportDir* property. To load ReportSmith Runtime and print the specified report, use the *Run* method (i.e., `Report1.Run`). The report prints on the default printer defined in ReportSmith. *Preview* is a Boolean property that specifies whether to print the report or just display it: If set to True, *Run* will display the report onscreen only; if set to False, *Run* will print the report.

The *AutoUnload* property specifies whether to automatically unload the ReportSmith Runtime executable after a report is run. Generally, if an application runs one report at a time, *AutoUnload* should be True. If an application is going to run a series of reports, then *AutoUnload* should be False.

The *InitialValues* property is of type *TStrings* and specifies the report variables to use with the report. Each line specifies a report variable as follows:

```
REPORTVAR = value
```

For more information about creating and using report variables, see *Creating Reports*.

Some important methods of *TReport* are listed in the following table:

Table 3.11 Important TReport methods

Method	Purpose
Run	Run a report.
RunMacro	Send a macro command to ReportSmith.

Table 3.11 Important TReport methods (continued)

Method	Purpose
Connect	Preconnect the report to a database, so it does not prompt for login.
SetVariable	Change a specific report variable.
ReCalcReport	Run a report again. Use this when report variables have changed.

Using TBatchMove

TBatchMove enables you to perform operations on groups of records or entire tables. The primary uses of this component are

- Downloading data from a server to a local data source for analysis or other operations.
- Upsizing a database from a desktop data source to a server. For more information on upsizing, see Appendix C, "Using local SQL."

TBatchMove is powerful because it can create tables on the destination that correspond to the source tables, automatically mapping the column names and data types as appropriate.

Two *TBatchMove* properties specify the source and a destination for the batch move operation: *Source* specifies a dataset (a *TQuery* or *TTable* component) corresponding to an existing source table. *Destination* specifies a *TTable* component corresponding to a database table. The destination table may or may not already exist.

Batch move modes

The *Mode* property specifies what the *TBatchMove* object will do:

Table 3.12 Batch move modes

Property	Purpose
batAppend	Append records to the destination table. The destination table must already exist. This is the default mode.
batUpdate	Update records in the destination table with matching records from the source table. The destination table must exist and must have an index defined to match records.
batAppendUpdate	If a matching record exists in the destination table, update it. Otherwise, append records to the destination table. The destination table must exist and must have an index defined to match records.
batCopy	Create the destination table based on the structure of the source table. The destination table must not already exist—if it does, the operation will delete it.
batDelete	Delete records in the destination table that match records in the source table. The destination table must already exist and must have an index defined.

The *Transliterate* property specifies whether to transliterate character data to the preferred character set for the destination table.

Data type mappings

In *Copy* mode, *TBatchMove* will create the destination table based on the column data types of the source table. In moving data between different table types, *TBatchMove* translates the data types as appropriate. The mappings from dBASE, Paradox, and InterBase data types are shown in the following tables.

Note To batch move data to an SQL server database, you must have that database server and Delphi Client/Server edition with the appropriate SQL Link installed. For more information, see the *SQL Links for Windows User's Guide*:

Table 3.13 Physical data type translations from Paradox tables to tables of other driver types

From Paradox type	To dBASE type	To Oracle type	To Sybase type	To InterBase type	To Informix type
Alpha	Character	Character	VarChar	Varying	Character
Number	Float {20.4}	Number	Float	Double	Float
Money	Float {20.4}	Number	Money	Double	Money {16.2}
Date	Date	Date	DateTime	Date	Date
Short	Number {6.0}	Number	SmallInt	Short	SmallInt
Memo	Memo	Long	Text	Blob/1	Text
Binary	Memo	LongRaw	Image	Blob	Byte
Formatted memo	Memo	LongRaw	Image	Blob	Byte
OLE	OLE	LongRaw	Image	Blob	Byte
Graphic	Binary	LongRaw	Image	Blob	Byte
Long	Number {11.0}	Number	Int	Long	Integer
Time	Character {>8}	Character {>8}	Character {>8}	Character {>8}	Character {>8}
DateTime	Character {>8}	Date	DateTime	Date	DateTime
Bool	Bool	Character {1}	Bit	Character {1}	Character
AutoInc	Number{11.0}	Number	Int	Long	Integer
Bytes	Memo	LongRaw	Image	Blob	Byte
BCD	N/A	N/A	N/A	N/A	N/A

Table 3.14 Physical data type translations from dBASE tables to tables of other driver types

From dBASE type	To Paradox type	To Oracle type	To Sybase type	To InterBase type	To Informix type
Character	Alpha	Character	VarChar	Varying	Character
Number	Short	Number	SmallInt	Short	SmallInt
others	Number	Number	Float	Double	Float
Float	Number	Number	Float	Double	Float
Date	Date	Date	DateTime	Date	Date
Memo	Memo	Long	Text	Blob/1	Text
Bool	Bool	Character {1}	Bit	Character {1}	Character
Lock	Alpha {24}	Character {24}	Character {24}	Character {24}	Character
OLE	OLE	LongRaw	Image	Blob	Byte

Table 3.14 Physical data type translations from dBASE tables to tables of other driver types (continued)

From dBASE type	To Paradox type	To Oracle type	To Sybase type	To InterBase type	To Informix type
Binary	Binary	LongRaw	Image	Blob	Byte
Bytes	Bytes	LongRaw	Image	Blob	Byte (only for temp tables)

Table 3.15 Physical data type translations from InterBase tables to tables of other driver types

From Interbase type	To Paradox type	To dBASE type	To Oracle type	To Sybase type	To Informix type
Short	Short	Number {6.0}	Number	Small Int	SmallInt
Long	Number	Number {11.0}	Number	Int	Integer
Float	Number	Float {20.4}	Number	Float	Float
Double	Number	Float {20.4}	Number	Float	Float
Char	Alpha	Character	Character	VarChar	Character
Varying	Alpha	Character	Character	VarChar	Character
Date	DateTime	Date	Date	DateTime	DateTime
Blob	Binary	Memo	LongRaw	Image	Byte
Blob/1	Memo	Memo	Long	Text	Text

By default *TBatchMove* matches columns 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. This is a list of column mappings (one per line) in one of two forms. To map the column, ColName, in the source table to the column of the same name in the destination table:

```
ColName
```

Or, to map the column named SourceColName in the source table to the column named DestColName in the destination table:

```
DestColName = SourceColName
```

If source and destination column data types are not the same, *TBatchMove* will perform a “best fit”. It will trim character data types, if necessary, and attempt 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, *TBatchMove* will convert a character value of ‘5’ to the corresponding integer value. Values that cannot be converted will generate errors. See “Handling batch move errors” on page 92.

Executing a batch move

Use the *Execute* method to execute the batch operation at run time. For example, if BatchMoveAdd is the name of a *TBatchMove* 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 *TBatchMove* component and choosing Execute. The *MoveCount* property keeps track of the number of records that were moved when a batch move is executed.

Handling batch move errors

There are basically 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 specify how it handles errors. The *AbortOnProblem* property specifies whether to abort the operation when a data type conversion error occurs. The *AbortOnKeyViol* property indicates whether to abort the operation when an integrity (key) violation occurs.

The following properties enable a *TBatchMove* 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 the batch operation.
- *KeyViolTableName*, if specified, creates a local (Paradox) table containing all records from the source table that caused an integrity violation (such as a key violation) as a result of the batch operation.
- *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.

Accessing the BDE directly

Delphi provides a wide range of built-in methods, properties, and functions that provide an interface to the Borland Database Engine, but applications are not limited to them. Some advanced applications may require direct access to BDE function calls, cursors, and so on. While direct BDE calls can provide additional functionality, they should be performed with caution, as they bypass Delphi's built-in functionality that keeps data-aware components synchronized with datasets.

If your application requires direct access to the BDE, you should first get the *Borland Database Engine User's Guide* from Borland. This documentation provides a complete reference and user's guide to the BDE.

The application must include the header files that reference the BDE: DBIPROCS.PAS and DBITYPES.PAS. Then the code can make direct calls to the BDE application programming interface.

BDE function calls often require parameters to specify the action to be performed. Delphi provides access to these through the following properties of dataset components:

- *DBHandle* is the handle for the database to which they are connected.
- *Handle* is the handle for the underlying cursor on the database.
- *DBLocale* and *Locale* are used for ANSI/OEM conversion for localization of applications.

After performing a BDE call directly, it is a good idea to call *Refresh* to ensure that all data-aware components are synchronized with their datasets.

Application examples

This section provides some brief examples of specific database tasks, illustrating some of the material presented in the preceding sections.

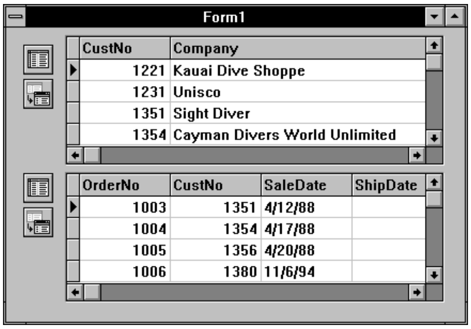
Creating a master-detail form

In the following example, you will create a simple form in which the user can scroll through customer records, and display all orders for the current customer. Follow these steps to create this application:

1. Place two *TTable*, two *TDataSource*, and two *TDBDataGrid* components on a form.
2. Set the properties of the first *TTable* component as follows:
   - *DatabaseName*: DBDEMO (the alias for the directory with the MAST database).
   - *TableName*: CUSTOMER (the table containing customer records).
   - *Name*: CustTable (for ease-of-use).
3. Name the first *TDataSource* component “CustDataSource,” and set its *Dataset* property to “CustTable.” Set the *DataSource* property of DBGrid1 to “CustDataSource.” When you activate CustTable (by setting its *Active* property to True), the grid displays the data in the CUSTOMER table.
4. Analogously, set the properties of the second *TTable* component as follows:
   - *DatabaseName*: DBDEMO.
   - *TableName*: ORDERS (the table containing order records).
   - *Name*: OrdTable (for ease-of-use).
5. Name the second *TDataSource* component “OrdDataSource,” and set its *Dataset* property to “OrdTable.” Set the *DataSource* property of DBGrid2 to “OrdDataSource.” When you activate OrdTable (by setting its *Active* property to True), the grid displays the data in the ORDERS table.

6 Compile and run the application now. The form displays data from each table independently and should look something like this:

Figure 3.8 Sample form



CustNo	Company
1221	Kauai Dive Shoppe
1231	Unisco
1351	Sight Diver
1354	Cayman Divers World Unlimited

OrderNo	CustNo	SaleDate	ShipDate
1003	1351	4/12/88	
1004	1354	4/17/88	
1005	1356	4/20/88	
1006	1380	11/6/94	

7 The next step is to link the ORDERS table (the master table) to the CUSTOMER table (the detail table) so that the form displays only the orders placed by the current customer. To do this, exit the application, return to design mode, and set the *MasterSource* property of OrdTable to CustDataSource.

8 In the Object Inspector, click on the ellipsis button to the right of the *MasterFields* property of OrdTable. The Field Link Designer dialog box will open.

- In the Available Indexes field, choose ByCustNo to link the two tables by the CustNo field.
- Select CustNo in both the Detail Fields and Master Fields field lists.
- Click on the Add button to add this join condition. In the Joined Fields list, “CustNo -> CustNo” will appear.
- Choose OK to commit your selections and exit the Field Link Designer.

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.

The *MasterSource* property specifies the *TDataSource* from which *OrdTable* will take its master column values. This limits the records it retrieves, based on the current record in CustTable. To do this, you must specify for *OrdTable*:

- The name of the column that links the two tables. The column must be present in each table, and must be identically named.
- The index of the column in the ORDERS table that will be linked to the CUSTOMER table.

In addition, you must ensure that the ORDERS table has an index on the *CustNo* field. Since it is a primary index, there is no need to specifically name it, and you can safely leave the *IndexName* field blank in both tables. However, if the table were linked through a secondary index, you must explicitly designate that index in the *IndexName* property.

In this example, the CUSTOMER table has a primary index on the *CustNo* column, so there is no need to specify the index name. However, the ORDERS table does not have a primary index on *CustNo*, so you must explicitly declare it in the *IndexName* property, in this case *ByCustNo*

Note You can also set the *IndexFieldNames* property to CustNo, and the correct index will be supplied for you.

Displaying multiple views of a table

Applications often need to display two different views of the same dataset. For instance, if you have two forms, and each displays different columns of the same row, you need to find a way to keep these forms synchronized to ensure accurate and up-to-date views of the data in each form.

There are two main concepts important to working with multiple views of a table:

- The *TDataSource* component provides access for multiple forms sharing a single dataset.
- When you want more than one different view of one table, put a *TDataSource* on each form that contains data aware components. You only need put a single *TTable* or *TQuery* component on the first form.

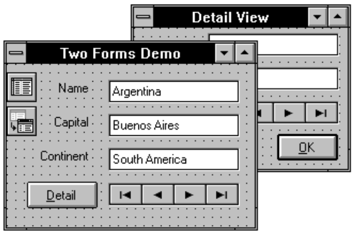
The simplest way to do this is to place a *TDataSource* on each form, and connect the first data source to the dataset you want to access. At run time you can then set the *DataSet* property of the second *TDataSource* to the *TTable* on the first form, for example:

```
Form2.DataSource2.Dataset := Form1.Table1;
```

Once the second *TDataSource* is assigned to a valid dataset, both forms will remain synchronized.

The TWOFORMS.DPR example in \DELPHI\DEMOS\DB\TWOFORMS demonstrates how to work with multiple forms and a single dataset. The program opens the COUNTRY table and shows the Name, Capital, and Continent fields on one form, and the Area and Population fields on a second form. A button on the first form opens the second form. Both forms have *TDBNavigator* components, so you can navigate through the table. The forms look like this:

Figure 3.9 Two forms



✔ To create the program manually,

1 Place a *TTable*, a *TDataSource*, a *TButton*, three *TDBEdit*, and three *TLabel* components on a form.

2 Give the Button the name "Detail."

3 Set the Table1's *DatabaseName* property to the DBDemos alias, and open the COUNTRY Table.

4 Connect DataSource1 to *Table1*, then connect each of the DBEdit component's *Datasource* properties to DataSource1. By performing these steps in this order, you can drop down a list in the *DataField* property of the data aware controls.

5 Create a second form, and place a data source, a *TBitBtn*, two *TDBLabels*, a *TDBNavigator* and two *TDBEdit* controls on the form. Don't yet connect the data source on this form to anything else.

6 Connect the DBEdit components to the DataSource, then enter 'Area' for the *DataField* property of DBEdit1, and 'Population' for DBEdit2.

7 Set the *Kind* property of the bitbutton to 'OK'.

8 Name this second form DetailView, then save the whole unit under the name DETAILS.PAS.

Note When you are creating the second form, you might find it helpful to temporarily create a *TTable* component and link the data source to it. This enables you to design the form using live data, and gives you access to a list of field names for each edit control. Once you have the form set up properly, you can delete the *TTable* component and at run time reconnect the data source to a table on a separate form.

✔ Once you have built the second form, go back to the first form and add DETAILS to the **uses** clause in the form unit's **implementation** part, and create the following event handler for the *OnClick* event of the Detail button:

```
procedure TForm1.DetailClick(Sender: TObject);
begin
  DetailView.DataSource1.DataSet := Table1;
  DetailView.Visible := True;
end;
```

This code first assigns the data source on the second form to the table in the first form. Once this connection is made, then the data source on the second form is "live." That is, it will act exactly like the data source on the first form. Then it makes the form visible.

✔ Now, connect the navigators on each form to the appropriate data source.

When you run the program, open the second form by clicking on the Detail button. Notice that whether you use the navigator in either form, the edit controls on the other form remain synchronized with the current record.

✔ Close the application and add two more lines of code to the second form's unit. This code is called in response to a click on the OK button:

```
procedure TDetailView.BitBtn1Click(Sender: TObject);
begin
  DataSource1.DataSet := nil;
  Close;
end;
```

This code sets the dataset to `nil` whenever the second form is closed. While not necessary, this ensures that the hidden detail view is not responding to events.

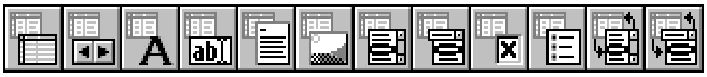
Chapter

4

Using Data Controls

To display and edit data from a database, use the components on the Data Controls page of the Component palette. Data controls include components such as *TDBGrid* for displaying and editing all specified records and fields in a table, and *TDBNavigator* for navigating among records, deleting records, and posting records when they change.

Figure 4.1 Data Controls Component palette



The following table summarizes the data controls in order from left to right as they appear on the Component palette:

Table 4.1 Data controls

Data control	Description
TDBGrid	Displays information from a data source in a spreadsheet-like grid. Columns in the grid can be specified at design time using the Fields Editor or at run time (dynamically bound).
TDBNavigator	Provides buttons for navigating through data obtained from a data source. At design time, you can choose to include one or more buttons to navigate through records, update records, post records, and refresh the data from the data source.
TDBText	Displays data from a specific column in the current data record.
TDBEdit	Uses an edit box to display and edit data from a specific column in the current data record.
TDBMemo	Displays memo-type database data. Memo fields can contain multiple lines of text or can contain BLOB (binary large object) data.
TDBImage	Displays graphic images and BLOB data from a specific column in the current data record.
TDBListBox	Displays a list of items from which a user can update a specific column in the current data record.
TDBComboBox	Combines a TDBEdit control with an attached list. The application user can update a specific column in the current data record by typing a value or by choosing a value from the drop-down list.

Table 4.1 Data controls (continued)

Data control	Description
TDBCheckBox	Displays a check box. If the value in the indicated column of the current record matches the text of the check box's *ValueChecked* property, the box is checked.
TDBRadioGroup	Offers a mutually exclusive set of options to enter into a specific column in the current data record in the form of radio buttons.
TDBLookupList	Displays a list of items from which a user can update a column in the current data record. The list of items is looked up in a specific column in another dataset.
TDBLookupCombo	Combines a TDBEdit control with a dropdown version of TDBLookupList. The application user can update a field in the current database by typing a value or by choosing a value from the drop-down list that is looked up in a specific column in another dataset.

Most data controls are data-aware versions of standard components. Some, such as *TDBGrid* and *TDBNavigator*, are data-aware, but differ from standard components in significant, useful ways. A *data-aware* control derives display data from a database source outside the application, and can also optionally post (or return) data changes to a data source. Data controls are data-aware at *design time*, meaning that when you connect a component to an active data source while building an application, you can immediately see live data in the controls. You can use the Fields Editor to scroll through records at design time to verify that your application is making the right database connections without requiring you to compile and run the application, but you cannot modify data at design time.

This chapter describes basic features common to all Data Control components, then describes how and when to use individual components.

Data Control component basics

Data controls are linked to database tables through the *DataSource* property. The *DataSource* property specifies the name of the *TDataSource* component from which a control gets its data. The *TDataSource* component is linked to a dataset (for example, *TTable* or *TQuery*) that is, in turn, connected to a database table. A dataset component is a grouping of *TField* components that can be dynamically created for you at run time, or statically specified using the Fields Editor at design time. For more information about *TDataSource*, *TTable*, *TQuery*, and the Fields Editor, see Chapter 3, "Using data access components and tools."

Data controls can only access columns in tables for which there are corresponding *TField* components. If the Fields Editor is used to limit a dataset to a subset of columns in a table, then *TField* components exist only for those columns. Most data controls provide a *DataField* property where you can specify the *TField* component with which it should be associated.

When designing a form that accesses data, you must place at least one dataset component (for example, *TTable* or *TQuery*), at least one *TDataSource* component, and one or more data controls on the form.

To place a data control on a form and link it to a dataset, follow these steps:

1 Drop the control on the form.

2 Set the *DataSource* property to the name of a *TTable* or *TQuery* component already on the form. You can type the name or choose it from the drop-down list.

3 Set the *DataField* property to the name of a *TField* component. You can type the field name or choose it from the drop-down list.

Note Two data controls, *TDBGrid* and *TDBNavigator*, access all available *TField* components within a dataset, and therefore do not have *DataField* properties. For these controls, omit step 3.

When a data control is associated with a dataset, its *Enabled* property determines if its attached *TDataSource* component receives data from mouse, keyboard, or timer events. Controls are also disabled if the *Enabled* property of *TDataSource* is False, or if the *Active* property of the dataset is False.

Updating fields

Most data controls can update fields by default. Update privileges depend on the status of the control's *ReadOnly* property *and* underlying *TField*'s and dataset's *CanModify* property. *ReadOnly* is set to False by default, meaning that data modifications can be made.

In addition, the data source must be in *Edit* state if updates are to be permitted. The data source *AutoEdit* property, set to True by default, ensures that the dataset enters Edit mode whenever an attached control starts to modify data in response to keyboard or mouse events.

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

Displaying data as labels with TDBText

TDBText is a read-only control similar to the *TLabel* component on the Standard page of the Component palette. *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*. When *TDBText* is linked to a data field at design time, however, you can see the current value for that field. For example, the following *TDBText* box, from the tutorial example, MASTAPP, displays company names.

Figure 4.2 TDBText component

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 zipcode field from a separate table. A *TDBText* component tied to the zipcode table could be used to display the zipcode field that matches the address entered by the user.

Note When you create 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 set to False, and the control is too small, data display is truncated.

Displaying and editing fields with TDBEdit

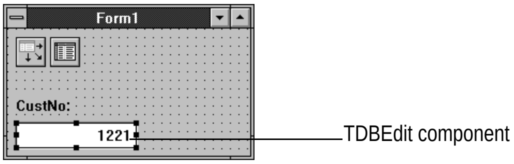
TDBEdit is a data-aware version of the *TEdit* component. *TDBEdit* displays the current value of a data field to which it is linked. You can also modify values in this component.

For example, suppose DataSource1 is a *TDataSource* component that is active and linked to an open *TTable* called Customer. You can then create a *TDBEdit* component (*DBEdit1*), and set its properties as follows:

- *DataSource*: DataSource1
- *DataField*: CUSTNO

The *DBEdit1* component immediately displays the value of the current row in the CUSTNO column of the CUSTOMER table, both at design time and at run time.

Figure 4.3 TDBEdit component at design time



Editing a field

A user can modify a database field in a *TDBEdit* component if:

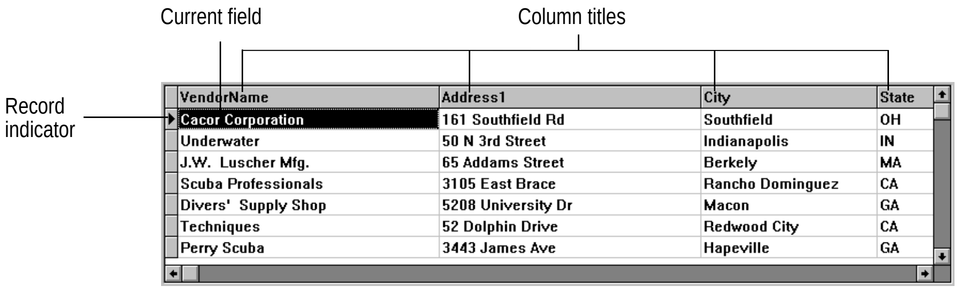
1. The *Dataset* is in Edit state.
2. The *Can Modify* property of the *Dataset* is True.
3. The *ReadOnly* property of *TDBEdit* is False.

Note Edits made to a field must be posted to the database by using a navigation or *Post* button on a *TDBNavigator* component.

Viewing and editing data with TDBGrid

TDBGrid enables you to view and edit all records associated with a dataset in a spreadsheet-like format:

Figure 4.4 TDBGrid component



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

The appearance of records in *TDBGrid* depends entirely on whether the *TField* components of the dataset are dynamically created by Delphi at run time, or if you use the Fields Editor to create persistent set of *TField* components whose properties you can specify in the Object Inspector at design time.

If Delphi generates a dynamic dataset at run time, then all records are displayed using default record and field ordering, and default display and edit formats. In most cases, however, you will want to control field order and appearance. To do so, use the Fields Editor to instantiate *TField* components and set their properties at design time.

When you use the Fields Editor to instantiate *TField* components, you gain a great deal of flexibility over the appearance of records in a *TDBGrid*. For example, the order in which fields appear from left to right in *TDBGrid* is determined by the way you order *TField* components in the Fields list box of the Fields Editor. Similarly, the *DisplayFormat* and *EditFormat* properties of a *TField* component determine how that field appears in *TDBGrid* during display and editing, respectively. You can also ensure that a value is entered for a field in a new record by setting its *Required* property to True. You can even prevent a *TField* component from being displayed in a grid by setting its *Visible* property to False. For more information about using the Fields Editor to control *TField* properties, see Chapter 3, "Using data access components and tools."

To put a *TDBGrid* on a form and link it to a dataset:

1 Create the control on the form.

2 Set the *DataSource* property to the name of a *TTable* or *TQuery* component already on the form.

Setting grid options

You can use the *TDBGrid Options* property at design time to control grid behavior and appearance at run time. When a *TDBGRid* 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, double-click the *Options* property. The list of options that you can set appears in the Object Inspector below the *Options* property. The + sign changes to a – (minus) sign, indicating that you can collapse the list of properties by double-clicking the *Options* property.

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

Table 4.2 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.

For more information about these options, see the online Help reference.

Editing in the grid

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

1 The *CanModify* property of the *Dataset* is True.
2 The *ReadOnly* property of *TDBGrid* is False.

In most data controls, edits to a field are posted as soon as you Tab to another control. By default in *TDBGrid*, edits and insertions within a field are posted only when you move to a different record in the grid. Even if you use the mouse to change focus to another control on a form, the grid does not post changes until you move off the current row. When a record is posted, Delphi checks all data-aware components 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 does not modify the record.

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

Rearranging column order at run time

At run time, 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*. Dragging a column to a new position in the grid affects the order of columns in the grid and the dataset. It does not change the order of columns in the underlying physical table.

To prevent rearrangement of column order, set the *DragMode* property to *dmAutomatic*.

Controlling grid drawing

You can control how individual cells in a grid are drawn. The *DefaultDrawing* property controls drawing of cells. By default, *DefaultDrawing* is set to True, meaning that Delphi uses the grid's default drawing method to paint cells and the data they contain. Data is drawn according to the *DisplayFormat* or *EditFormat* properties of the *TField* component associated with a given column.

If you set a grid's *DefaultDrawing* property to False, Delphi does not draw cells or cell contents in the grid. You must supply your own drawing routine that is keyed to the *OnDrawDataCell* event for *TDBGrid*.

Using events to control grid behavior

You can modify grid behavior by writing events handlers to respond to specific actions within the grid. 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 *TDBGrid* events available in the Object Inspector:

Table 4.3 TDBGrid events

Event	Purpose
OnColEnter	Specify action to take when a user moves into a column on the grid.
OnColExit	Specify action to take when a user leaves a column on the grid.
OnDblClick	Specify action to take when a user double clicks in the grid.
OnDragDrop	Specify action to take when a user drags and drops in the grid.
OnDragOver	Specify action to take when a user drags over the grid.

Table 4.3 TDBGrid events (continued)

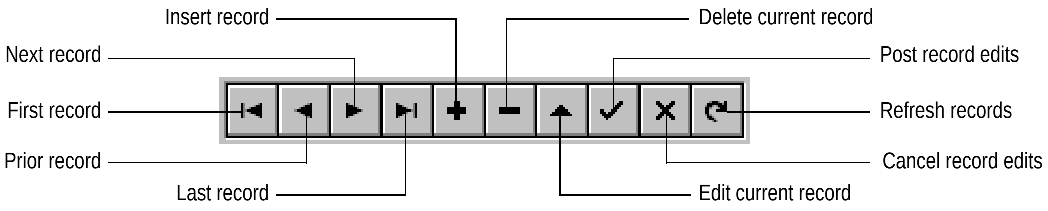
Event	Purpose
OnDrawDataCell	Customize drawing of grid cells.
OnEndDrag	Specify action to take when a user stops dragging on the grid.
OnEnter	Specify action to take when the grid gets focus.
OnExit	Specify action to take when the grid loses focus.
OnKeyDown	Specify action to take when a user presses any key or key combination on the keyboard when in the grid.
OnKeyPress	Specify action to take when a user presses a single alphanumeric key on the keyboard when in the grid.
OnKeyUp	Specify action to take when when a user releases a key when in the grid.

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. For more information about *TDBGrid* events and properties, see the online Help reference.

Navigating and manipulating records with TDBNavigator

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 4.5 TDBNavigator component



The following table describes the buttons on the navigator:

Table 4.4 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.

Table 4.4 TDBNavigator buttons (continued)

Button	Purpose
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.

Hiding and showing navigator buttons

When you first put a *TDBNavigator* on a form, 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, on a form that is intended for browsing rather than editing, you might want to disable the *Edit*, *Insert*, *Delete*, *Post*, and *Cancel* buttons.

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, double-click the *VisibleButtons* property. 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, indicating that you can collapse the list of properties by double-clicking the *VisibleButtons* property.

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 and run times.

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.

For more information about buttons and the methods they call, see the online Help reference.

Displaying fly-by help

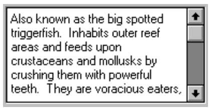
By default, the navigator displays fly-by Help Hints whenever you pass the mouse cursor over the navigator buttons. The *ShowHint* property of *TDBNavigator, set to* False *by default,* controls this functionality. Set *ShowHints* to True to display Help Hints.

You can provide your own hint text for a navigator by entering separate strings for each hint in the *Hints* property. The strings you enter override the default hints provided by the navigator control.

Displaying and editing BLOB text with TDBMemo

TDBMemo is a data-aware component—similar to the Standard *TMemo* component—that can display binary large object (BLOB) data. *TDBMemo* displays multi-line text, and permits a user to enter multi-line text as well. For example, you can use *TDBMemo* controls to display memo fields from dBASE and Paradox tables and text data contained in BLOB fields.

Figure 4.6 TDBMemo component



Also known as the big spotted
triggerfish. Inhabits outer reef
areas and feeds upon
crustaceans and mollusks by
crushing them with powerful
teeth. They are voracious eaters,

By default, *TDBMemo* permits a user to edit memo text. To limit the number of characters users can enter into the database memo, use the *MaxLength* property. To make a *TDBMemo* component read-only, set its *ReadOnly* property to True.

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. To permit tabs in a memo, set the *WantTabs* property to True. The *Alignment* property determines how the text is aligned within the control. Possible choices are *taLeftJustify* (the default), *taCenter*, and *taRightJustify*.

At run time, 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.

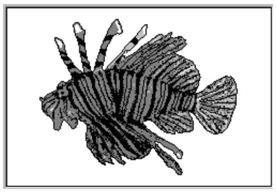
Because the *TDBMemo* can display large amounts of data, it can take time to populate the display at run time. To reduce the time it takes scroll through data records, *TDBMemo* has an *AutoDisplay* property that controls whether the accessed data should automatically displayed. If you set *AutoDisplay*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

Note A *TDBMemo* control raises an exception if an attempt is made to access fields that contain more than 32K of data, or if edited data exceeds 32K in length.

Displaying BLOB graphics with TDBImage

TDBImage is a data-aware component that displays bitmapped graphics contained in BLOB data fields. It captures BLOB graphics images from a dataset, and stores them internally in the Windows .DIB format.

Figure 4.7 DBImage component



By default, *TDBImage* permits a user to edit a graphics image by cutting and pasting to and from the Clipboard, or if you supply an editing method. You can accomplish the same task programmatically by using the *CutToClipboard*, *CopyToClipboard*, and

PasteFromClipboard methods. To make a *TDBImage* component read-only, set its *ReadOnly* property to True.

Because the *TDBImage* can display large amounts of data, it can take time to populate the display at run time. 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.

Using list and combo boxes

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

Note These components can be linked only to *TTable* components. They do not work with *TQuery* components.

The following table describes these controls:

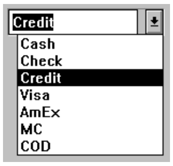
Table 4.5 Data-aware list box and combo box controls

Data control	Description
TDBListBox	Displays a list of items from which a user can update a specific column in the current data record.
TDBComboBox	Combines a TDBEdit control with an attached list. The application user can update a specific column in the current data record by typing a value or by choosing a value from the drop-down list.
TDBLookupList	Displays a list of items from which a user can update a column in the current data record. The list of items is looked up in a specific column in another dataset.
TDBLookupCombo	Combines a TDBEdit control with a dropdown version of TDBLookupList. The application user can update a field in the current database by typing a value or by choosing a value from the drop-down list that is looked up in a specific column in another dataset.

TDBComboBox

The *TDBComboBox* component is similar to a *TDBEdit* component, except that at run time it has a drop-down list that enables a user to pick from a predefined set of values. Here is an example of what a *TDBComboBox* component looks like at run time:

Figure 4.8 DBComboBox component



The *Items* property of the component specifies the items contained in the drop-down list. Use the String List Editor to specify the values for the drop-down list.

At run time, the user can choose an item from the list or (depending on the value of the *Style* property) type in a different entry. When the component is linked to a column through its *DataField* property, it displays the value in the current row, regardless of whether it appears in the *Items* list.

The following properties determine how the *Items* list is displayed at run time:

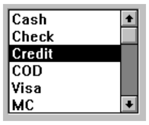
- *Style* determines the display style of the component:
  - *csDropDown* (default): Displays a drop-down list with an edit box in which the user can enter text. All items are strings and have the same height.
  - *csSimple*: Displays the *Items* list at all times instead of in a drop-down list. All items are strings and have the same height.
  - *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 run time.
  - *csOwnerDrawFixed* and *csOwnerDrawVariable*: Allows the items list to display values other than strings (for example, bitmaps). For more information, see the online VCL reference.
- *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 will be displayed in alphabetical order.

TDBListBox

TDBListBox is functionally the same as *TDBComboBox*, but instead of a drop-down list, it displays a scrollable list of available choices. When the user selects a value at run time, the component is assigned that value. Unlike *TDBComboBox*, the user cannot type in an entry that is not in the list.

Here is an example of how a *TDBListBox* component appears at run time.

Figure 4.9 TDBListBox component



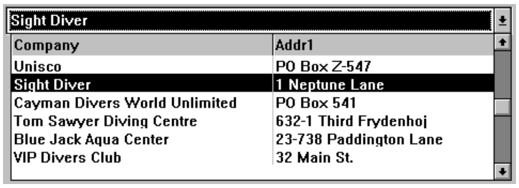
While navigating through a dataset, a *TDBListBox* component displays values in the column by highlighting the corresponding entry in its list. If the current row's value is not defined in the *Items* property, no value is highlighted in the *TDBListBox*. Changing the selection changes the underlying value in the database column and is equivalent to typing a value in a *TDBEdit* component.

The *IntegralHeight* property controls the way the list box is displayed. If *IntegralHeight* is True (the default), the bottom of the list box moves up to the bottom of the last completely-displayed item in the list. If *IntegralHeight* is False, the bottom of the list box is determined by the *ItemHeight* property, and the bottom item might not be completely displayed.

TDBLookupCombo

The *TDBLookupCombo* component is similar to *TDBComboBox*, except that it derives its list of values dynamically from a second dataset at run time, and it can display multiple columns in its drop-down list. With this control, you can ensure that users enter valid values into forms by providing an interface from which they can choose values. Here is an example of how a *TDBLookupCombo* component appears at run time:

Figure 4.10 TDBLookupCombo component



Sight Diver

Company	Addr1
Unisco	PO Box Z-547
Sight Diver	1 Neptune Lane
Cayman Divers World Unlimited	PO Box 541
Tom Sawyer Diving Centre	632-1 Third Frydenhoj
Blue Jack Aqua Center	23-738 Paddington Lane
VIP Divers Club	32 Main St.

The lookup list for *TDBLookupCombo* must be derived from a second dataset. To display values from a column in the same table as the first dataset, drop a second data source and dataset component on the form and point them at the same data as the first data source and dataset.

Three properties establish the lookup list for *TDBLookupCombo*, and determine how it is displayed:

- *LookupSource* specifies a second data source from where the control populates its list.
- *LookupField* specifies a field in the *LookupSource* dataset which links that dataset to the primary dataset. This must be a column in the dataset pointed to by *LookupSource*, and it must contain the same values as the column pointed to by the *DataField* property (although the column names do not have to match).
- *LookupDisplay*, if set, defines the columns that *TDBLookupCombo* displays. If you do not specify values in *LookupDisplay*, *TDBLookupCombo* displays the values found in the column specified by *LookupField*. Use this property to display a column other than that specified by *LookupField*, or to display multiple columns in the drop-down list. To specify multiple columns, separate the different column names with a semicolon.

A *TDBLookupCombo* component appears similar to a *TDBComboBox* at both design time and run time, except when you want it to display multiple columns in its lookup list. How the control displays multiple columns depends on the *Options* property settings:

- *loColLines*: When True, uses lines to separate the columns displayed in the lookup list.
- *loRowLines*: When True, uses lines to separate the rows displayed in the lookup list.
- *loTitles*: When True, column names appear as titles above the columns displayed in the lookup list.

As a simple example, an order entry form could have a *TDBLookupCombo* component to specify the customer number of the customer placing the order. The user placing the order can simply click on the drop down “pick list” instead of having to remember the customer number. The value displayed could be the customer name.

To build this form,

1 Choose File | New Project to create a new form.

2 Create a *TDataSource* component onto the form, then set its *Name* property to “OrdSource.”

3 Drop a *TTable* component on the form, and set the *Name* property to “OrdTable,” the *DatabaseName* property to “DBDEMOS,” the *TableName* property to “ORDERS.DB,” and the *Active* property to True.

4 Create a second *TDataSource* component on the form, then set its *Name* property to “CustSource.”

5 Create a second *TTable* component on the form, and set the *Name* property to “CustTable,“ the *DatabaseName* property to “DBDEMOS,” the *TableName* property to “CUSTOMER.DB,” and the *Active* property to True.

6 Create a *TDBGrid* component and link it to OrdSource so it displays the contents of the ORDERS table.

7 Create a *TDBLookupCombo* component, and set its *DataSource* property to “CustNo.” The database lookup combo box is now linked to the CustNo column of the ORDERS table.

8 Specify the lookup values of the *TDBLookupCombo* component:
   - Set *LookupSource* to “CustSource” (so it looks up values in the CUSTOMER table).
   - Set *LookupField* to “CustNo” (so it looks up and gets values from the CustNo column).
   - In *LookupDisplay*, type `Company;Addr1` (this displays the corresponding company name and address in the drop-down list).

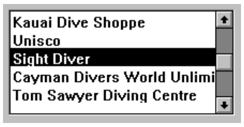
9 Set the *loColLines* and *LoTitles* properties (under the *Option* property) of the *TDBLookupCombo* to True.

At run time, the *TDBLookupCombo* component displays a drop-down list of company names and addresses. If the user selects a new company from the list, the control is assigned the value of the corresponding customer number (CustNo). When the user scrolls off the current order in the database grid, Delphi posts the new customer number and information to the row.

TDBLookupList

TDBLookupList is functionally the same as *TDBLookupCombo*, but instead of a drop-down list, it displays a scrollable list of the available choices. When the user selects one at run time, the component is assigned that value. Like *TDBLookupCombo*, the user cannot type in an entry that is not in the list. Here is an example of how a *TDBLookupList* component appears at run time:

Figure 4.11 TDBLookupList component


Kauai Dive Shoppe
Unisco
Sight Diver
Cayman Divers World Unlimi
Tom Sawyer Diving Centre

While navigating through a dataset, a *TDBLookupList* component highlights the item in the list that corresponds to the value in the currently selected row. If the current row's value is not defined in the *Items* property, no value is highlighted in the *TDBLookupList* component. Changing the selection changes the underlying value in the database column and is equivalent to typing a value in a *TDBEdit* component.

TDBCheckBox

TDBCheckBox is a data-aware version of the Standard *TCheckBox* component. It can be used to set the values of fields in a dataset. For example, a customer invoice form might have a check box control that when checked, specifies that the customer is entitled to a special discount, or when unchecked means that the customer is not entitled to a discount

Figure 4.12 TDBCheckBox component

Like the other data controls, *TDBCheckBox* is attached to a specific field in a dataset through its *DataSource* and *DataField* properties. Use the *Caption* property to display a label for the check box on your form.

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.

If the *DataField* of the check box 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.

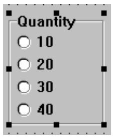
A *TDBCheckBox* component is grayed out and disabled whenever the field for the current record does not contain one of the values listed in the *ValueChecked* or *ValueUnchecked* properties.

TDBRadioGroup

TDBRadioGroup is a data-aware version of the standard *TRadioGroup* component. It lets you 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 consists of one button for each value a field can accept.

TDBRadioGroup is attached to a specific field in a dataset through its *DataSource* and *DataField* properties. A radio button for each string value entered in the *Items* property is displayed on the form, and the string itself is displayed as a label to the right of the button.

Figure 4.13 A TDBRadioGroup component



For the current record, if the field associated with a radio group matches one of the strings in the *Items* or 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 is 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, to continue the example for the three buttons labeled “Red,” “Yellow,” and “Blue,” if three strings, “Magenta,” “Yellow,” and “Cyan,” are listed for *Values*, and the user selects the first button (labeled “Red”), then Delphi posts “Magenta” to the database.

If strings for *Values* are not provided, the label from a selected radio button (from *Items*) is returned to the database when a record is posted. Users can modify the value of a data field by clicking the appropriate radio button. When the user scrolls off the current row, Delphi posts the value indicated by the radio button string to the database.

Chapter

5

Using SQL in applications

SQL (Structured Query Language) is an industry-standard language for database operations. Delphi enables your application to use SQL syntax directly through the *TQuery* component. Delphi applications can use SQL to access data from:

- Paradox or dBASE tables, using local SQL. The allowable syntax is a sub-set of ANSI-standard SQL and includes basic SELECT, INSERT, UPDATE, and DELETE statements. For more information on local SQL syntax, see Appendix C, "Using local SQL."
- Databases on the Local InterBase Server. Any statement in InterBase SQL is allowed. For information on syntax and limitations, see the *InterBase Language Reference*.
- Databases on remote database servers (Delphi Client/Server only). You must have installed the appropriate SQL Link. Any standard statement in the server's SQL is allowed. For information on SQL syntax and limitations, see your server documentation.

Delphi also supports heterogeneous queries against more than one server or table type (for example, data from an Oracle table and a Paradox table). For more information, see "Creating heterogenous queries" on page 124.

Using TQuery

TQuery is a dataset component, and shares many characteristics with *TTable*, as described in Chapter 3, "Using data access components and tools." In addition, *TQuery* enables Delphi applications to issue SQL statements to a database engine (either the BDE or a server SQL engine).

The SQL statements can be either static or dynamic, that is, they can be set at design time or include parameters that vary at run time.

When to use TQuery

For simple database operations, *TTable* is often sufficient and provides portable database access through the BDE. However, *TQuery* provides additional capabilities that *TTable* does not. Use *TQuery* for:

- Multi-table queries (joins).
- Complex queries that require sub-SELECTs.
- Operations that require explicit SQL syntax.

TTable does not use SQL syntax; *TQuery* uses SQL, which provides powerful relational capabilities but may increase an application's overall complexity. Also, use of non-standard (server-specific) SQL syntax may decrease an application's portability among servers; for more information, see Chapter 6, "Building a client/server application."

How to use TQuery

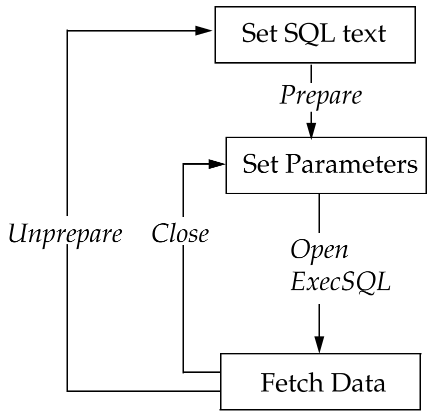
To access a database, set the *DatabaseName* property to a defined BDE alias, a directory path for desktop database files, or a file name for a server database. If the application has a *TDatabase* component, *DatabaseName* can also be set to a local alias that it defines. For more information, see "Using the TDatabase component" in Chapter 6, "Building a client/server application."

To issue SQL statements with a *TQuery* component:

- Assign the *TQuery* component's *SQL* property the text of the SQL statement. You can do this:
  - At design time, by editing the *TQuery*'s SQL property in the Object Inspector, choosing the *SQL* property, and entering the SQL statements in the String List Editor dialog box. With Delphi Client/Server, you can also use the Visual Query Builder to construct SQL syntax.
  - At run time, by closing any current query with *Close*, clearing the *SQL* property with *Clear*, and then specifying the SQL text with the *Add* method.
- Execute the statement with the *TQuery* component's *Open* or *ExecSQL* method. Use *Open* for SELECT statements. Use *ExecSQL* for all other SQL statements. The differences between *Open* and *ExecSQL* are discussed in a subsequent section.
- To use a dynamic SQL statement, use the *Prepare* method, provide parameters and then call *Open* or *ExecSQL. Prepare* is not required, but will improve performance for dynamic queries executed multiple times.

The following diagram illustrates the lifetime of a *TQuery* component and the methods used to work with it:

Figure 5.1 TQuery methods and flow



Note *Prepare* applies only to dynamic queries. It is not required, but is recommended in most cases. For more information, see "Dynamic SQL statements" on page 121.

The SQL property

The *SQL* property contains the text of the SQL statement to be executed by a Query component. This property is of type *TStrings*, which means that it is a series of strings in a list. The list acts very much as if it were an array, but it is actually a special class with unique capabilities. For more information on *TStrings*, see the online VCL reference.

A Query component can execute two kinds of SQL statements:

- Static SQL statements
- Dynamic SQL statements

A *static* SQL statement is fixed at design time and does not contain any parameters or variables. For example, this statement is a static SQL statement:

```
SELECT * FROM CUSTOMER WHERE CUST_NO = 1234
```

A *dynamic* SQL statement, also called a *parameterized* statement, includes parameters for column or table names. For example, this is a dynamic SQL statement:

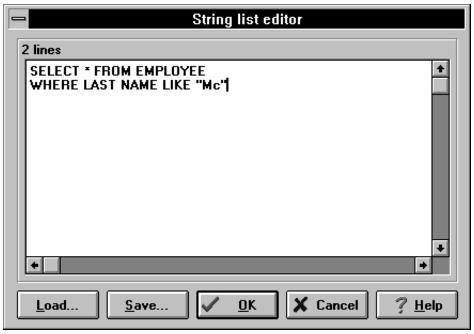
```
SELECT * FROM CUSTOMER WHERE CUST_NO = :Number
```

The variable *Number*, indicated by the leading colon, is a parameter which must be provided at run time and may vary each time the statement is executed.

Creating the query text

You can enter the SQL text for a *TQuery* at design time by double-clicking on the *SQL* property in the Object Inspector, or choosing the ellipsis button. The String List Editor opens, enabling you to enter an SQL statement.

Figure 5.2 Editing SQL statements in the String List Editor



```
SELECT * FROM EMPLOYEE
WHERE LAST NAME LIKE "Mc"
```

Choose OK to assign the text you enter to the *SQL* property of the query. Choose Load to include text from a file or Save to save the text to a file.

To specify SQL text at run time, an application should first close the query with *Close* and clear the *SQL* property with *Clear*. For example,

```
Query1.Close; {This closes the query}
Query1.SQL.Clear; {This clears the contents of the SQL property}
```

It is always safe to call *Close*—if the query is already closed, the call will have no effect. Use the *SQL* property's *Add* method to add the SQL statements to it. For example,

```
Query1.SQL.Add('SELECT * FROM COUNTRY');
Query1.SQL.Add('WHERE NAME = ''ARGENTINA''');
```

An application should always call *Clear* before specifying an SQL statement. Otherwise, *Add* will simply append the statements to the existing one.

Note The *SQL* property may contain only one complete SQL statement at a time. In general, multiple statements are not allowed. Some servers support multiple statement "batch" syntax; if the server supports this, then such statements are allowed.

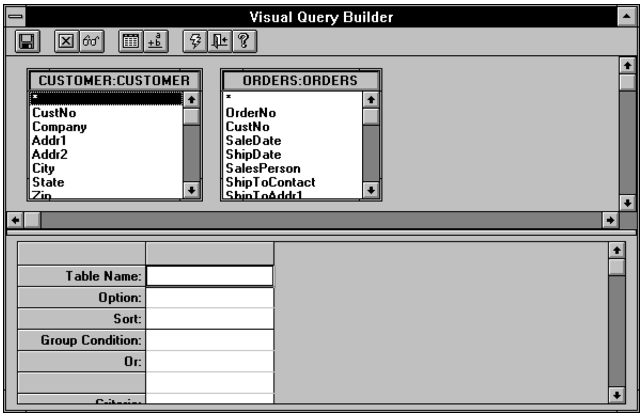
You can also use the *LoadFromFile* method to assign the text in an SQL script file to the *SQL* property. For example,

```
Query1.SQL.LoadFromFile('C:\MYQUERY.TXT');
```

Using the Visual Query Builder

Delphi Client/Server includes a Visual Query Builder that enables you to construct SQL SELECT statements visually. To invoke the Visual Query Builder, right click on a *TQuery* component and select Run Visual Query Builder. A dialog box prompts you to select the database to work with; select the desired database and choose OK. Another dialog box will prompt you to enter the tables you want to query; select the desired tables, choosing Add after each, and then choose Close. The Visual Query Builder window will then become active with the select tables.

Figure 5.3 Working in the Visual Query Builder



For information on how to use the Visual Query Builder, refer to its online Help. After you have you constructed a query and exited the Visual Query Builder, the SQL statement you constructed will be entered in the *SQL* property of the selected *TQuery* component.

Executing a query

At design time, you can execute a query by changing its *Active* property in the Object Inspector to True. The results of the query will be displayed in any data controls connected to the Query component (through a data source).

At run time, an application can execute a query with either the *Open* or the *ExecSQL* methods. Use *Open* for SQL statements that return a result set (SELECT statements). Use *ExecSQL* for all other SQL statements (INSERT, UPDATE, DELETE, and so on). For example,

```
Query1.Open; {Returns a result set}
```

If the SQL statement does not return a cursor and a result set from the database, use *ExecSQL* instead of *Open*. For example,

```
Query1.ExecSQL; {Does not return a result set}
```

If you don't know at design time whether a query will return a result set, use a **try...except** block with *Open* in the **try** part and *ExecSQL* in the **except** part.

The UniDirectional property

Use the *UniDirectional* property to optimize access to a database table through a *TQuery* component. If you set *UniDirectional* to True, you can iterate through a table more quickly, but you will only be able to move in a forward direction. *UniDirectional* is False by default.

Getting a live result set

A *TTable* component always returns a live result set to an application. That is, the user sees the data "live" from the database, and can make changes to it directly through data controls. A *TQuery* can return two kinds of result sets:

- "Live" result sets: As with *TTable* components, users can edit data in the result set with data controls. The changes are sent to the database when a *Post* occurs, or when the user tabs off a control, as described in Chapter 4, "Using Data Controls."
- "Read only" result sets: Users cannot edit data in the result set with data controls.

By default, a query always returns a read-only result set. To get a live result set, an application must request it by setting the *RequestLive* property of *TQuery* to True. However, for the BDE to be able to return a live result set, the SELECT syntax of the query must conform to the guidelines given below. If an application requests a live result set, but the syntax does not conform to the requirements, the BDE returns a read-only result set (for local SQL) or an error return code (for passthrough SQL). If a query returns a live result set, Delphi will set the *CanModify* property to True.

Table 5.1 Types of query result sets

RequestLive	CanModify	Type of result set
False	False	Read-only result set
True—SELECT syntax meets requirements	True	Live result set
True—SELECT syntax does not meet requirements	False	Read-only result set

If an application needs to update the data in a read-only result set, it must use a separate *TQuery* to construct an UPDATE statement. By setting the parameters of the update query based on the data retrieved in the first query, the application can perform the desired update operation.

Syntax requirements for live result sets

To return a live result set, a query must have *RequestLive* set to True. The SQL syntax must conform to that of Local SQL, as described in Appendix C, "Using local SQL." Additionally, the syntax must meet the requirements described below.

A query of a Paradox or dBASE table can return a live result set if it:

- Involves only a single table.
- Does not have an ORDER BY clause.
- Does not use aggregates such as SUM or AVG.
- Does not use calculated fields in the SELECT list.
- The WHERE clause may consist only of comparisons of column names to scalar constants. The comparison operators may be LIKE, >, <, >=, and <=. The clause may contain any number of such comparisons linked by AND or OR operators.

A query of a server table using passthrough SQL can return a live result set if it:

- Involves a single table.

- Does not have an ORDER BY clause.
- Does not use aggregates such as SUM or AVG.

In addition, if the table is on a Sybase server, it must have a unique index.

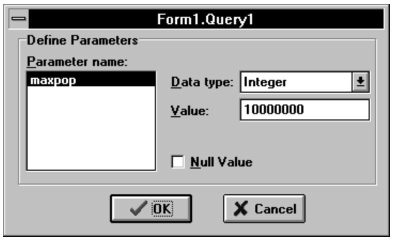
Dynamic SQL statements

A dynamic SQL statement (also called a parameterized query) contains parameters that can vary at run time.

Supplying values to parameters

At design time, you can supply values to parameters with the Parameters Editor. Invoke the Parameters Editor by selecting a *TQuery* component, right-clicking the mouse, and then selecting Parameters Editor. The Parameters Editor opens.

Figure 5.4 Parameters Editor



maxpop
Data type: Integer
Value: 10000000
Null Value

Select the desired data type for the parameter in the Data Type combo box. Enter a value in the Value text field or select Null Value to set the parameter's value to null. When you click OK, the query will be prepared and values will be bound to the parameters. Then, when you set the query's *Active* property to True, the results of the SQL query with the specified parameter values will be shown in any data controls connected to the query.

At run time, an application can supply values to parameters with the following *TQuery* properties:

- The *Params* property, using the order that the parameters appear in the SQL statement.
- The *ParamByName* method, using the parameter names specified in the SQL statement.
- The *DataSource* property to set values from another dataset for columns that match the names of parameters that have no values.

Preparing a query

The *Prepare* method sends a parameterized query to the database engine for parsing and optimization. A call to *Prepare* is not required to use a parameterized query. However, it is strongly recommended, since it will improve performance for dynamic queries that will be executed more than once. If a query is not explicitly prepared, each time it is executed, Delphi automatically prepares it.

Prepare is a Boolean property of *TQuery* that indicates if a query has been prepared. The Parameters Editor automatically prepares a query when you use it to set parameter values at design time.

If a query has been executed, an application must call *Close* before calling *Prepare* again. Generally, an application should call *Prepare* once—for example, in the *OnCreate* event of the form—then set parameters using the *Params* property, and finally call *Open* or *ExecSQL* to execute the query. Each time the query is to be executed with different parameter values, an application must call *Close*, set the parameter values, and then execute the query with *Open* or *ExecSQL*.

Preparing a query consumes some database resources, so it is good practice for an application to unprepare a query once it is done using it. The *UnPrepare* method unprepares a query. When you change the text of a query at run time, Delphi automatically closes and unprepares the query.

Using the Params property

When you enter a query, Delphi creates a *Params* array for the parameters of a dynamic SQL statement. *Params* is a zero-based array of *TParam* objects with an element for each parameter in the query; that is, the first parameter is *Parms*[0], the second *Params*[1], and so on.

For example, suppose a *TQuery* component named *Query2* has the following statement for its *SQL* property:

```
INSERT
  INTO COUNTRY (NAME, CAPITAL, POPULATION)
  VALUES (:Name, :Capital, :Population)
```

An application could use *Params* to specify the values of the parameters as follows:

```
Query2.Params[0].AsString := 'Lichtenstein';
Query2.Params[1].AsString := 'Vaduz';
Query2.Params[2].AsInteger := 420000;
```

These statements would bind the value "Lichtenstein" to the :Name parameter, "Vaduz" to the :Capital parameter, and 420000 to the :Population parameter.

Using the ParamByName method

ParamByName is a function that enables an application to assign values to parameters based on their names. Instead of providing the ordinal location of the parameter, you must supply its name.

For example, an application could use *ParamByName* could specify values for the parameters in the preceding example as follows:

```
Query2.ParamByName('Name').AsString := 'Lichtenstein';
Query2.ParamByName('Capital').AsString := 'Vaduz';
```

```
Query2.ParamByName('Population').AsInteger := 420000;
```

These statements would have the same effect as the three previous statements that used the *Params* array directly.

Using the DataSource property

For parameters of a query not bound to values at design time, Delphi will check the query's *DataSource* property. This property specifies the name of a *TDataSource* component. If *DataSource* is set, and the unbound parameter names match any column names in the specified *DataSource*, Delphi binds the current values of those fields to the corresponding parameters. This capability enables applications to have linked queries.

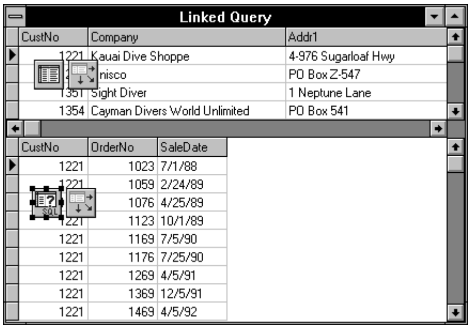
The LINKQRY sample application illustrates the use of the *DataSource* property to link a query in a master-detail form. The form contains a *TQuery* component (named Orders) with the following in its *SQL* property:

```
SELECT Orders.CustNo, Orders.OrderNo, Orders.SaleDate
   FROM Orders
   WHERE Orders.CustNo = :CustNo
```

As illustrated below, the form also contains:

- A *TDataSource* named OrdersSource, linked to Orders by its *DataSet* property.
- A *TTable* component (named Cust).
- A *TDataSource* named CustSource linked to Cust.
- Two data grids; one linked to CustSource and the other to OrdersSource.

Figure 5.5 Form with linked queries



Linked Query

CustNo	Company	Addr1
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy
[illegible]	nisco	PO Box Z-547
1351	Sight Diver	1 Neptune Lane
1354	Cayman Divers World Unlimited	PO Box 541

CustNo	OrderNo	SaleDate
1221	1023	7/1/88
1221	1059	2/24/89
[illegible]	1076	4/25/89
1221	1123	10/1/89
1221	1169	7/5/90
1221	1176	7/25/90
1221	1269	4/5/91
1221	1369	12/5/91
1221	1469	4/5/92

Orders' *DataSource* property is set to *CustSource*. Because the parameter :*CustNo* does not have any value assigned to it, at run time Delphi will try to match it with a column name in *CustSource*, which gets its data from the Customer table through *Cust*. Because there is a CustNo column in *Cust*, the current value of CustNo in the *Cust* table is assigned to the parameter, and the two data grids are linked in a master-detail relationship. Each time the *Cust* table moves to a different row, the *Orders* query automatically re-executes to retrieve all the orders for the current customer.

Dynamic SQL example

Here's a simple form that uses a dynamic query and provides substitution parameters programmatically:

- Start a new project and place the required controls on the form. For this example, place an edit control, Edit1, a button, Button1, and a data-aware grid control, DBGrid1, on the form. Now create a *TQuery* component, Query1, and a data source, DataSource1, and set the *DataSource* property of DBGrid1 to "DataSource1" and the *Dataset* property of DataSource1 to "Query1."

  Set the *DatabaseName* property of Query1 to DBDEMOS. Double-click on its SQL property in the Object Inspector and enter the following SQL statement. Remember to precede the substitution parameter with a colon:

  ```
  SELECT * FROM country WHERE name LIKE :CountryName
  ```

- Prepare the query in the *OnCreate* event of the form:

  ```
  procedure TForm1.FormCreate(Sender: TObject);
  begin
    Query1.Prepare;
  end;
  ```

- Provide parameters in response to some event. In this example, double-click on Button1 to edit the *OnClick* event and use the contents of Edit1.Text as a substitution parameter:

  ```
  procedure TForm1.Button1Click(Sender: TObject);
  begin
    Query1.Close;
    Query1.Params[0].AsString := Edit1.Text;
    Query1.Open;
  end;
  ```

Creating heterogenous queries

Some applications may require queries of tables in more than one database. Such queries are called *heterogenous queries*, and are not supported by standard SQL. A heterogenous query may join tables on different servers, and even different types of servers. For example, a heterogeneous query might involve a table in a Oracle database, a table in a Sybase database, and a local dBASE table.

Delphi supports heterogeneous queries, as long as the query syntax conforms to the requirements of local SQL, as described in Appendix C, "Using local SQL."

To perform a heterogeneous query, you must define a BDE standard alias that references a local directory, and use the alias for the *DatabaseName* of the query component. You must also define BDE aliases for each of the databases being queried. In the query text, precede each table name with the alias for its database.

You can define BDE aliases with the BDE Configuration Utility, described in Appendix B, “Using the BDE configuration utility.” 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, :Sybase1:ORDERS
```

Chapter

6

Building a client/server application

Delphi Client/Server enables you to develop applications that can access remote SQL servers such as Oracle, Sybase, Informix, and InterBase, as well as local Paradox and dBASE databases. A *remote* server is one that is physically removed from the client machine on which the Delphi application runs. The server and client must be connected by a network. Delphi also includes the Local InterBase Server, a full-featured SQL server that runs on Microsoft Windows.

There are a number of issues that are particularly important when developing a client/server application:

Portability versus optimization: Will the application use any server-specific SQL syntax? To what degree will the database be optimized for a particular server?

Transactions: What kind of transaction control will the application require?

Server features: Will the application require the use of server features such as stored procedures? How will these be surfaced?

Connectivity: What communication protocol will the application use? Does the application need to be deployed to support multiple communication protocols?

Deployment: What executables, libraries, and other files does the application require and how are these delivered to the end user?

Portability versus optimization

In a client/server system, the database running on the server and the application running on the client define the overall system, referred to as the *database/application.* While these two elements are often designed separately and considered distinct, they must be integrated to build a successful client/server application. One of the important considerations is portability versus optimization.

Portability refers to the ease with which an database/application can run on different servers. *Optimization* refers to the extent to which an application takes advantage of the special features of a particular system.

Client portability is not an issue with Delphi, because Delphi applications will run on any 16-bit or 32-bit Windows platform. However, server portability and communications portability can be considerations.

Because Delphi applications use the Borland Database Engine, they can be easily integrated with dBASE and Paradox applications (for desktop data sources) and other clients for server data sources.

Server portability

It may be desirable to design an application so that it can be easily ported to different types of servers, either because the end-users require multiple heterogeneous server support, or because the application will be used by different groups of end-users with different types of servers. In designing a client-server application, there is an inherent trade-off between portability and optimization, because making use of server-specific features results in increased application performance but decreased portability.

A Delphi application that uses only *TTable* components for data access will be fully portable among different server types. An application may benefit from improved performance by using *TQuery* components and passthrough SQL, and as long as the SQL syntax is ANSI standard, there will be little loss of portability.

As soon as SQL syntax departs from the ANSI standard, the application will no longer be fully portable. If server portability is a consideration, you must carefully weigh whether the gain in using server-specific syntax is worth the cost in portability. Maintainability of an application may be reduced by optimization for a specific server type, because each server-specific implementation may require separate maintenance.

An application can be further optimized by using server-specific features such as stored procedures. However, this will usually require server-specific implementation in the database, and perhaps the application, depending on how the features are surfaced.

It is also important to consider that servers' transaction processing may differ in subtle yet important ways. This and other distinctions among SQL servers may complicate portability. Before attempting to create a portable database/application, you should build an application that runs reliably against one type of server database. In some cases, it may be necessary to build the application separately against each of the target server types.

Client/server communication portability

Depending on the application requirements, it may be necessary to support multiple communication protocols, such as TCP/IP and Novell SPX. Providing for multiple communication protocols is simply a matter of ensuring that the client platforms have the proper communication software installed. This portability issue does not typically surface until the deployment phase, but it should be addressed in the implementation phase to ensure that the initial test deployment packages include the proper client communication software.

Using the Local InterBase Server

The Local InterBase Server (LIBS) is a version of the Borland InterBase server that runs on Windows 3.1. It provides all of the features of a SQL server for local, single-user operation. For more information on the Local Interbase Server, see the *Local InterBase Server User's Guide*. The Local Interbase Server can be used in primarily two ways for client/server development:

- As a local environment for developing a client/server application, regardless of the server to be used.
- As an intermediate step in upsizing, between the desktop and server, providing a local SQL engine for development of SQL-specific features. For more information on upsizing, see "Upsizing" on page 140.
- As a local database engine for deployment of standalone desktop SQL applications.

Building an application to access any server

If you are developing a client/server application, you can use the Local InterBase Server for local development, even if the production database will run on some other server type.

If the production database already exists on a database server, you can use the Local InterBase Server as follows:

- Use your server's tools to create an SQL data definition script for the database. Remove any non-standard SQL syntax that will not work with InterBase. Generally, this means removing stored procedure definitions and other advanced features and mapping data types to InterBase data types.
- Use Windows ISQL to create a local InterBase database then execute the SQL script to define the database. For information on using Windows ISQL, see the *Local InterBase Server User's Guide*.
- Populate the database with representative data using the techniques described in "Upsizing."
- Create a Delphi database application that uses the Local InterBase Server database as a data source.
- Once the application has been sufficiently tested against the development database, change the *TDataSource* to use an alias that points to the target server. It is recommended to first test the application against a duplicate of the production database to surface server-specific features such as stored procedures.

If the production database does not already exist, you can first define a prototype database on the LIBS, and develop the application locally. Simultaneously, you can define a congruent database on the target server. Finally, you can redirect the application to access the database on the target server.

Building an application to access InterBase

To build an application that accesses a production database that already exists on an InterBase workgroup server, follow this procedure:

- Use the InterBase database backup utility to create a backup of the database, using the "Backup Metadata Only" option. This will save the structure of the database, but not the data (which may be huge). For information on how to do this, see the *Local InterBase Server User's Guide*.
- Restore the database to the Local InterBase Server. This will be the development platform.
- Populate the database with a modest amount of representative data. Your application will access this data during the development and testing process. Because it is "dummy" data that is not connected to the production database, there is no chance that the production database will be corrupted. Your application can access any server features present in the production database, including stored procedures.
- Once the application has been sufficiently tested against the development database, change the *TDataSource* to use an alias that refers to the production server. It is recommended to first test the application against a duplicate of the production database on the server.

Using InterBase in upsizing

One upsizing path is to use the Local Interbase Server (LIBS) as an intermediate data source. This enables you to address all the SQL development issues separately from connectivity and client/server issues. The steps include

- Defining the database on the Local InterBase Server, using the techniques described in "Upsizing" on page 140.
- Developing the application against the LIBS database.
- Migrating the LIBS database to the target server.
- Redirecting the application to access the target server.

Connecting to a database server

Borland SQL Links for Windows enables a Delphi application to connect through the BDE to a remote database server. SQL Links drivers provide connections to Oracle, Sybase, Informix, Microsoft SQL Server, and InterBase.

Through the BDE Configuration Utility, you can set up an alias for each data source to which your application needs to connect. These aliases then become available to choose as the value of the *DatabaseName* property of *TTable* and *TQuery* components. For more information on the BDE Configuration Utility, see Appendix B, "Using the BDE configuration utility."

Connectivity

Delphi client applications can use any network protocol (such as TCP/IP or Novell SPX/IPX) supported by the server, as long as both the server and the client machines have the proper communication software installed. You must configure the SQL Link driver for the desired protocol. For more information, see the *SQL Links for Windows User's Guide*.

Establishing an initial connection between client and server can often be problematic, especially when using TCP/IP, because there are a number of critical factors that must all be in place before a connection can be established.

Using TCP/IP

TCP/IP is a widely-used communication protocol that enables applications to connect to many different database servers. When using TCP/IP, you must ensure:

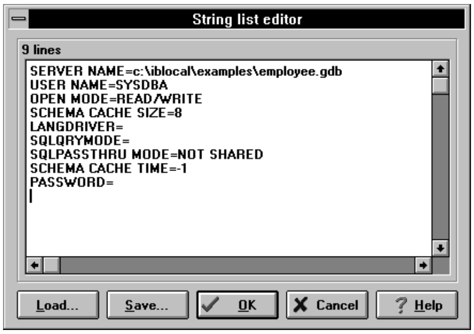
- The TCP/IP communication software and the proper Winsock driver are installed on the client.
- The server's IP address is registered in the client's HOSTS file or that Directory Network Services (DNS) is properly configured.
- The server's port number is entered in the client's SERVICES file.
- The application is searching the proper directory paths for the DLLs it needs. Check the PATH statement in AUTOEXEC.BAT.

For more information, see the *SQL Links for Windows User's Guide* and your server documentation.

Connection parameters

The *Params* property of a connected *TDatabase* object contains a *TString* list of all the SQL Link parameters required to connect to a server of the specified type. You can edit these parameters by clicking on the ellipsis button to the right of the *Params* property in the Object Inspector. The String List Editor opens with the parameters For example, here are the parameters for connection to an InterBase server:

Figure 6.1 InterBase parameters in the String List Editor



```
SERVER NAME=c:\iblocal\examples\employee.gdb
USER NAME=SYSDBA
OPEN MODE=READ/WRITE
SCHEMA CACHE SIZE=8
LANGDRIVER=
SQLQRYMODE=
SQLPASSTHRU MODE=NOT SHARED
SCHEMA CACHE TIME=-1
PASSWORD=
```

You can modify these parameters and add others as needed to customize the connection performed by the application. For more information, see the *SQL Links for Windows User's Guide*.

Using ODBC

A Delphi application can access ODBC data sources such as DB2, Btrieve, or Microsoft Access through the Borland Database Engine (BDE). To do this, you must set up an ODBC driver connection using the BDE Configuration Utility. An ODBC driver connection requires:

- A vendor-supplied ODBC driver.
- The Microsoft ODBC Driver Manager.
- A BDE alias, established with the BDE Configuration Utility or with Delphi.

The BDE configuration setting AUTO ODBC (on the System page) enables an alias to automatically configure for use of ODBC. When AUTO ODBC is True, datasource and driver information will automatically be imported from the ODBC.INI file.

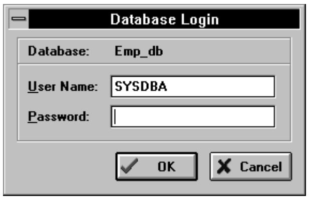
For more information, see the online help for the BDE Configuration Utility.

Handling server security

Most database servers include security features to limit database access. Generally, the server will require a user name and password login before a user can access a database. If a server requires a login, then Delphi will prompt you at design time when you attempt to connect to a database on the server (for example, when you specify a *TableName* for a *TTable* component).

By default, a Delphi application opens the standard Login dialog box, whenever an application opens a connection to a database server. If a connection has already been established, the Login dialog box does not appear.

Figure 6.2 Database Login dialog box



A Delphi application can handle server login several different ways:

- If the *LoginPrompt* property of a *TDatabase* component is True (the default), the standard Delphi Login dialog box will be opened when the application attempts to establish a database connection.
- By setting *LoginPrompt* to False, and including the USERNAME and PASSWORD parameters in the *Params* property of the *TDatabase* component. For example,

  ```
  USERNAME = SYSDBA
  PASSWORD = masterkey
  ```

  This is generally not recommended since it compromises server security.
- Use the *OnLogin* event of *TDatabase* to set login parameters. The *OnLogin* event gets a copy of the *TDatabase*'s login parameters array. Use the *Values* property to change these parameters:

  ```
  LoginParams.Values['SERVER NAME'] := 'MYSERVERNAME';
  LoginParams.Values['USER NAME'] := 'MYUSERNAME';
  LoginParams.Values['PASSWORD'] := 'MYPASSWORD';
  ```

  When control returns from your *DatabaseLogin* event handler, these parameters will be used to establish a connection.

Using the TDatabase component

The *TDatabase* component is not required for database access, but it provides additional control over factors that are important for client/server applications, including the ability to:

- Create a persistent database connection.
- Customize database server logins.
- Create BDE aliases local to an application.
- Control transactions and specify transaction isolation level.

The *DataSets* property of *TDatabase* is an array of pointers to the active datasets in the *TDatabase*. The *DatasetsCount* property is an integer that specifies the number of active datasets.

The Connected property

The *Connected* property of a *TDatabase* component specifies whether an application remains connected to a database server even when no tables are open. If an application will be opening and closing several tables in a single database, it will be more efficient to set the Database object's *Connected* property to True. That way, the application will remain connected to the database even if it does not have any tables open. It can then open and close tables repeatedly without incurring the overhead of connecting to the database each time.

A *TDatabase* object's *Connected* property can be overridden by the application-global *TSession* object. This object has a Boolean *KeepConnections* property that specifies whether to maintain database connections even if no tables are open. If *KeepConnections* is False, a *TDatabase* object's *Connected* property determines if connections are maintained when no *TTables* or *TQueries* are open. If *KeepConnections* is True (the default), database connections are always maintained. Specifically,

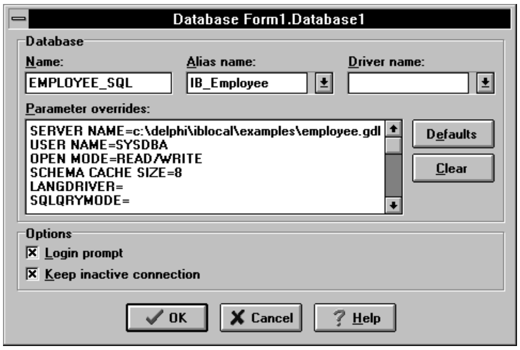
- Database connections will be maintained until the application exits or until the Session's *DropConnections* method is called .
- Setting a *TDatabase* object's *Connected* property to False will have no effect.

Creating application-specific aliases

The *TDatabase* object enables you to create BDE aliases specific to an application. To name the alias, enter a name in the *DatabaseName* property. Any dataset components can then use the local alias by using the specified *DatabaseName*.

To customize the parameters for a local alias, double-click on the *TDatabase* component. The Database Properties Editor opens:

Figure 6.3 Database Properties Editor


Database Form1.Database1

Database

Name: EMPLOYEE_SQL

Alias name: IB_Employee

Driver name:

Parameter overrides:

```
SERVER NAME=c:\delphi\iblocal\examples\employee.gdl
USER NAME=SYSDBA
OPEN MODE=READ/WRITE
SCHEMA CACHE SIZE=8
LANGDRIVER=
SQLQRYMODE=
```

Options

☒ Login prompt

☒ Keep inactive connection

This tool enables you to customize application-specific aliases local based on existing aliases.

The three text fields at the top of the dialog box correspond to the *DatabaseName*, *AliasName*, and *DriverName* properties.

- *DatabaseName* is the name of the database connection that can be used by dataset components instead of a BDE alias, directory path, or database name. In other words, this is the name of the local alias defined by the component that will show up in the *DatabaseName* drop-down list of dataset components.
- *AliasName* is the name of an existing BDE alias configured with the BDE Configuration Utility. This is where the component gets its default parameter settings. This property will be cleared if *DriverName* is set.
- *DriverName* is the name of a BDE driver, such as STANDARD (for dBASE and Paradox), ORACLE, SYBASE, INFORMIX or INTERBASE. This property will be cleared if *AliasName* is set, because an *AliasName* specifies a driver type.

Choose the Defaults button to retrieve the default parameters for the selected alias. The values will be displayed in the Parameters list box. Any changes you make to the defaults are used instead of the default values for any database connection in the application that uses that *DatabaseName*.

The check boxes labeled "Loginprompt" and "Keep inactive connection" correspond to the *LoginPrompt* and *KeepConnections* properties of the *TDatabase* component.

Understanding transaction control

SQL database servers handle requests in logical units of work called *transactions*. A transaction is a group of SQL statements that must all be performed successfully before the server will finalize (or *commit*) changes to the database. Either all the statements will succeed, or all will fail.

Transaction processing ensures database consistency even if there are hardware failures, and maintains the integrity of data while allowing concurrent multiuser access. For example, an application might update the ORDERS table to indicate that an order for a certain item was taken, and then update the INVENTORY table to reflect the reduction in inventory available. If there were a hardware failure after the first update but before the second, the database would be in an inconsistent state, since the inventory would not reflect the order entered. Under transaction control, both statements would be committed at the same time. Transaction control becomes even more important in a multiuser application.

In SQL, transactions are explicitly ended with a command to either accept or discard the actions performed. The COMMIT statement permanently commits the transaction, making changes visible to all users. The ROLLBACK statement undoes all changes made to the database in the transaction. Different database servers implement transaction processing differently. For the specifics of how your server handles transaction processing, refer to your server documentation.

Handling transactions in applications

Delphi applications can control transactions:

- **Implicitly:** Delphi will automatically start and commit transactions as needed when an application calls the *Post* method (explicitly or implicitly in another method).

- **Explicitly:** depending on the level of control the application requires, either with
  - The *StartTransaction*, *Commit*, and *Rollback* methods of *TDatabase.* This is the recommended approach.
  - Passthrough SQL in a *TQuery* component. The application must use server-specific SQL transaction control statements. You must understand how your server performs transaction handling.

Transaction control statements are only meaningful when the database is on an SQL server. The *StartTrans*, *Commit*, and *Rollback* methods will raise an exception if the underlying database is Paradox or dBASE.

Implicit transaction control

Delphi applications that use only the built-in methods can rely on implicit transaction control. Any Delphi operations on a server database that are not under explicit transaction control will be under implicit control. Delphi will commit each individual write operation (*Post*, *AppendRecord*, and so on) as a separate transaction, so it will commit database changes on a row-by-row basis. This minimizes update conflicts, but can lead to heavy network traffic.

When using implicit transaction control, keep the SQLPASSTHRUMODE setting at SHARED AUTOCOMMIT, the default. For more information, see "Setting the SQL passthrough mode."

Implicit transaction control happens automatically, but does not provide much flexibility. If an application needs multi-row transactions or passthrough SQL, then it should use explicit transaction control.

Explicit transaction control

The recommended approach for transaction control is to use the methods of *TDatabase*, because this will result in clearer code and a more portable application. The methods for transaction control are

- *StartTransaction*: Begins a transaction at the isolation level specified by the *TransIsolation* property of *TDatabase*. If a transaction is currently active, Delphi will raise an exception.
- *Commit*: Commits the currently active transaction on the database. If no transaction is active, Delphi will raise an exception.
- *Rollback*: Rolls back the currently active transaction. All changes to the database since the last *Commit* will be undone.

Some applications may require additional server-specific transaction control features. In this case, use a *TQuery* component with passthrough SQL statements for transaction control. Ensure that SQLPASSTHRUMODE is set to NOT SHARED so that the passthrough SQL does not affect other transactions.

Setting the SQL passthrough mode

SQLPASSTHRUMODE in the BDE Configuration utility determines if passthrough SQL and standard BDE calls share the same database connection. For transactions, this translates to whether passthrough transactions and other transactions "know" about each other. Only applications that use passthrough SQL need be concerned with SQLPASSTHRUMODE.

SQLPASSTHRUMODE can have the following settings:

- SHARED AUTOCOMMIT (the default)
- SHARED NOAUTOCOMMIT
- NOT SHARED

With SHARED AUTOCOMMIT, each operation on a single row is committed. This mode most closely approximates desktop database behavior, but is inefficient on SQL servers because it starts and commits a new transaction for each row, resulting in a heavy load of network traffic.

With SHARED NOAUTOCOMMIT, the application must explicitly start and commit transactions. This setting can result in conflicts in busy, multiuser environments where many users are updating the same rows.

NOT SHARED means that passthrough SQL and Delphi methods use separate database connections.

Note To control transactions with passthrough SQL, you must set SQLPASSTHRU MODE to NOT SHARED. Otherwise, passthrough SQL and Delphi's methods may interfere with each other, leading to unpredictable results.

Transaction isolation levels

A transaction's *isolation level* determines how it interacts with other simultaneous transactions accessing the same tables. In particular, the isolation level affects what a transaction reads from the tables being accessed by other transactions.

Some servers enable you to set the transaction isolation level explicitly in passthrough SQL. If not specified, passthrough SQL operations will use a server's default isolation level. For more information, see your server documentation.

Transactions (both explicit and implicit) using Delphi's built-in methods will use the *TransIsolation* property of *TDatabase* to specify transaction isolation level. *TransIsolation* can have the following values:

- *tiDirtyRead*: The transaction can read uncommitted changes to the database by other transactions. This is the lowest isolation level.
- *tiReadCommitted*: The transaction can read only committed changes to the database by other transactions. This is the default isolation level.
- *tiRepeatableRead*: The transaction cannot read other transactions' changes to previously read data. This guarantees that once a transaction reads a records, it will not change if it reads it again. This the highest isolation level.

Database servers may support these isolation levels differently or not at all. If the requested isolation level is not supported by the server, then Delphi will use the next highest isolation level. The actual isolation level used by each type of server is shown in

Table 6.1, "Server transaction isolation levels." For a detailed description of how each isolation level is implemented, see your server documentation.

Table 6.1 Server transaction isolation levels

TransIsolation setting	Oracle	Sybase and Microsoft SQL servers	Informix	InterBase
Dirty read	Read committed	Read committed	Dirty Read	Read committed
Read committed (Default)	Read committed	Read committed	Read committed	Read committed
Repeatable read	Repeatable read (READ ONLY)	Read committed	Repeatable Read	Repeatable Read

If an application is using ODBC to interface with a server, the ODBC driver must also support the isolation level. For more information, see your ODBC driver documentation.

Using stored procedures

A stored procedure is a server-based program that can take input parameters and return output parameters to an application. Stored procedures are associated with a database, and are actually part of metadata, like tables or domains. The *TStoredProc* component enables Delphi applications to execute server stored procedures.

The *DatabaseName* property of *TStoredProc* specifies the database in which the stored procedure is defined. This property is the same as for *TTable* and *TQuery*--it can be a BDE alias or an explicit directory path and database name. The *StoredProcName* specifies the name of the stored procedure. A drop-down list will display a list of all procedures defined in the specified database.

A *TStoredProc* can return either a singleton result or a result set with a cursor, if the server supports it.

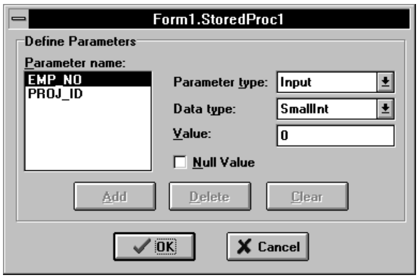
Note InterBase "select" procedures are called with the SELECT statement as if querying a table. To get output from such procedures, use a *TQuery* component with the appropriate SELECT syntax.

Input and output parameters

A stored procedure has a *Params* array for its input and output parameters similar to a *TQuery* component. The order of the parameters in the *Params* array is determined by the stored procedure definition. An application can set the values of input parameters and get the values of output parameters in the array similar to *TQuery* parameters. You can also use *ParamByName* to access the parameters by name. If you are not sure of the ordering of the input and output parameters for a stored procedure, use the Parameters Editor.

To invoke the Parameters Editor, select the *TStoredProc* component and then right-click the mouse. The following dialog box opens:

Figure 6.4 TStoredProc Parameters Editor



The Parameters Editor displays the input and output parameters for the procedure. To prepare the stored procedure with the default parameter types and field types, simply choose OK. You can set values of input parameters at design time by choosing the parameter in the Parameters list and entering a value in the Value field. To specify null input parameter values, select the Null value check box. The Parameters Editor is explained in more detail in Chapter 5, "Using SQL in applications."

Note Delphi will attempt to get information on input and output parameters from the server. For some servers (such as Sybase), this information may not be accessible. In such cases, you must enter the names and data types of the input and output parameters in the Parameters Editor at design time.

Executing a stored procedure

Before an application can execute a stored procedure, you must prepare the stored procedure, which can be done:

- At design time with the Parameters Editor.
- At run time with the *Prepare* method of *TStoredProc*.

To prepare a stored procedure at run time, use the *Prepare* method, before executing it. For example,

```
StoredProc1.Prepare;
```

To execute a prepared stored procedure, use the *ExecProc* method. Values can be assigned to and from a *TStoredProc* component just as for *TQuery* components, by using the *Params* array. For example, the following code could be in a button's *OnClick* event:

```
StoredProc1.Params[0].AsString := Edit1.Text;
StoredProc1.ExecProc;
Edit2.Text := StoredProc1.Params[1].AsString;
```

The first parameter, *Params*[0], is an input parameter of type String. It is assigned the text entered by the user in Edit1. Then, assuming StoredProc1 has been prepared at design time with the Parameters Editor, the stored procedure is run with *ExecProc*. Finally, the output parameter, *Params*[1], is displayed by Edit2.

On some servers, stored procedures can return a result set similar to a query. Applications can use data aware controls to display the output of such stored procedures. You do this in the same way as you display output from *TQuery* components: create a *TDataSource* component and assign its name to a data grid's *DataSource* property.

Oracle overloaded stored procedures

Oracle servers allow overloading of stored procedures; that is, different procedures with the same name. The *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 Delphi will execute the first stored procedure with the overloaded name; if it is two (2), it will execute the second, and so on.

Upsizing

Migrating a desktop application to a client/server application is called *upsizing*. Upsizing is a complex topic and a full treatment of it is beyond the scope of this book. However, this section will address some of the most important aspects of upsizing a Delphi application.

Upsizing has two major facets:

- Upsizing the database from the desktop to the server
- Upsizing the application to address client/server considerations

Upsizing requires a shift in perspective from the desktop world to the client/server world. Desktop databases and SQL server databases are different in many respects. Desktop databases are designed for one user at a time, while servers are designed for multiuser access. Desktop databases are conceptually record-oriented, while server databases are conceptually set-oriented. Desktop databases typically store each table in a separate file, while servers store all the tables in a database together.

Client/server applications must also address some entirely new issues, the most complex of which are connectivity, network usage, and transaction handling.

Upsizing the database

Upsizing a database includes the following steps:

- Defining metadata on the server, based on the existing desktop database structure.
- Migrating the data from the desktop to the server.
- Addressing issues such as:
  - Data type differences
  - Data Security and Integrity
  - Transaction control
  - Data Access Rights
  - Data Validation
  - Locking

Delphi provides two ways to upsize a database:

- Use the Database Desktop utility and choose Tools | Utilities | Copy to copy a table from desktop table to SQL format. For more information, see Appendix A, "Using Database Desktop."
- Build a Delphi application that uses a *TBatchMove* component. For more information on *TBatchMove*, see Chapter 3, "Using data access components and tools."

Both of these options will copy table structures and migrate data from the desktop source to the server destination. Depending on the database, it may be necessary to make changes to the tables created by these methods. For example, the datatype mappings may not be exactly as desired.

Additionally, you must add to the database any of the following features if required:

- Integrity constraints (primary and foreign keys)
- Indexes
- Check constraints
- Stored procedures and triggers
- Other server-specific features

Depending on the database, it may be most efficient to define the metadata first by using an SQL script and the server's data definition tools and then migrate the data using one of the two methods previously mentioned. If you define the table structure manually, then Database Desktop and *TBatchMove* will copy only the data.

Upsizing the application

In principle, a Delphi application designed to access local data can access data on a remote server with few changes to the application itself. If a congruent data source has been defined on an SQL server, you can re-direct the application to access it rather than the local data source, simply by changing the *DatabaseName* property of *TTable* or *TQuery* components in the application.

In practice, however, there are a number of important differences between accessing local and remote data sources. Client/server applications must also address a number of issues that are not relevant to desktop applications.

Any Delphi application can use either *TTable* or *TQuery* for data access. Desktop applications will generally use the *TTable* component. When upsizing to a SQL server, it may be more efficient to use *TQuery* objects instead in some instances. Depending on the specific application, *TQuery* may be preferable if the application will be retrieving a large number of records from database tables.

If the application uses mathematical or aggregate functions, it may be more efficient to perform these functions on the server with stored procedures. The use of stored procedures may be faster because servers are typically more powerful. This also reduces the amount of network traffic required, particularly for functions that process a large number of rows.

For example, an application might need to compute the standard deviation of values of a large number of records. If this function were performed on the client, all the values would have to be retrieved from the server to the client, resulting in a lot of network traffic. If the function were performed by a stored procedure, all the computation would be performed on the server, so the application would only retrieve the answer from the server.

Deploying support for remote server access

Deployment of general database application is discussed in Chapter 1, "Introduction." In addition to the files required to deploy a desktop database application, deployment of a client/server application requires installation of the appropriate Borland SQL Links. These are not part of the Borland Database Engine, and must be installed separately. They are redistributable, according to the terms of the license agreement.

Note For more information on deploying Delphi applications, refer to the file DEPLOY.TXT, installed to the DELPHI\DOC directory by default.

Each server type has a set of files for the SQL Link. In addition, a file used by all the SQL Links is BLROM800.LD, the Roman8 language driver using binary collation sequence.

Oracle

The following files provide the SQL Links interface with Oracle servers. In addition, applications will require Oracle client files for interface to low-level communication protocols such as TCP/IP. Refer to your server documentation.

Table 6.2 Oracle SQL Link files

File name	Description
SQLD_ORA.DLL	Borland SQL Link Oracle Driver
SQLD_ORA.HLP	Online help file
SQL_ORA.CNF	BDE Configuration File for Oracle Driver
ORA6WIN.DLL	Oracle Version 6.x client-side DLL
ORA7WIN.DLL	Oracle Version 7.x client-side DLL
SQL13WIN.DLL	Oracle client-side DLL
SQLWIN.DLL	Oracle client-side DLL
COREWIN.DLL	Oracle client-side DLL
ORAWE850.LD	Language driver based on DOS code page 850

Sybase and Microsoft SQL servers

The following files provide the SQL Links interface with Sybase servers. In addition, applications will require Sybase client files for interface to low-level communication protocols such as TCP/IP. Refer to your server documentation.

Table 6.3 Sybase SQL Link files

File name	Description
SQLD_SS.DLL	Borland SQL Link Sybase Driver
SQLD_SS.HLP	Borland SQL Link Sybase Driver Help
SQL_SS.CNF	BDE Configuration File for Sybase Driver

Table 6.3 Sybase SQL Link files (continued)

File name	Description
W3DBLIB.DLL	Sybase/Microsoft SQL Server client-side DLL
DBNMP3.DLL	Sybase/Microsoft SQL Server client-side DLL for Named Pipes
SYDC437.LD	Language driver based on DOS code page 850
SYDC850.LD	Language driver based on DOS code page 437

Informix

The following files provide the SQL Links interface with Informix servers. In addition, applications will require Informix client files for interface to low-level communication protocols such as TCP/IP. Refer to your server documentation.

Table 6.4 Informix SQL Link files

File name	Description
SQLD_INF.DLL	Borland SQL Link Informix Driver
SQLD_INF.HLP	Online help file
SQL_INF.CNF	BDE Configuration File for Informix Driver
LDLLSQLW.DLL	Informix client-side DLL
ISAM.IEM	Informix error message file
OS.IEM	Informix error message file
RDS.IEM	Informix error message file
SECURITY.IEM	Informix error message file
SQL.IEM	Informix error message file

InterBase

The following files provide the SQL Links interface to remote InterBase servers. These files are distinct from those required to access the Local InterBase Server..

Table 6.5 InterBase SQL Link files

File name	Description
SQLD_IB.DLL	Borland SQL Link InterBase Driver
SQLD_IB.HLP	Borland SQL Link InterBase Driver Help
SQL_IB.CNF	BDE Configuration File for InterBase Driver
CONNECT.EXE	InterBase connection diagnostic tool
CONNECT.HLP	InterBase Windows connection diagnostic help file
GDS.DLL	InterBase API DLL
REMOTE.DLL	InterBase Networking interface DLL
INTERBAS.MSG	InterBase error message file

TCP/IP Interface

The following files provide InterBase client applications their interface to Winsock 1.1 compliant TCP/IP products.

Table 6.6 Winsock 1.1 client files

File name	Description
MVWASYNC.EXE	Asynchronous communication module
VSL.INI	TCP/IP transport initialization file
WINSOCK.DLL	Windows Socket DLL
MSOCKLIB.DLL	Maps Windows socket calls to VSL driver

For TCP/IP products that are not Winsock 1.1 compliant, InterBase client applications will require one of the following files. During installation, Delphi will prompt you to select the TCP/IP stack for which to install support. If the deployed application needs to support a different TCP/IP stack, you must copy the corresponding file from the installation disks.

Table 6.7 Non-Winsock compliant TCP support files

File name	TCP/IP Product
M3OPEN.EXE	3Com 3+Open TCP Microsoft LAN Manager Digital Pathworks for DOS
M3OPEN.DLL	3Com 3+Open TCP Version 2.0
MBW.EXE	Beame & Whiteside TCP/IP
MFTP.EXE	FTP PC/TCP
MHPARPA.DLL	HP ARPA Service for DOS
MNETONE.EXE	Ungermann-Bass Net/One
MNOVLWP.DLL	Novell LAN WorkPlace for DOS
MPATHWAY.DLL	Wollongong Pathway Access for DOS
MPCNFS.EXE	Sun PC NFS
MPCNFS2.EXE	Sun PC NFS v3.5
MPCNFS4.DLL	Sun PC NFS v4.0
MWINTCP.EXE	Wollongong WIN TCP\IP for DOS

Other communication protocols

The InterBase workgroup server for NetWare supports Novell SPX/IPX protocol. Two client files are required: NWIPXSPX.DLL and NWCALLS.DLL.

The InterBase Workgroup Server for Windows NT supports Microsoft Named Pipes protocol. No additional client files are required to support Named Pipes, but the client machine must have Microsoft LAN Manager or Windows for Workgroups 3.1.1 installed.

Deploying ReportSmith support

To deploy an application that uses a *TReport*, you must include the .RPT files and ReportSmith Runtime in your deployment package. By default, Delphi installs the files required for ReportSmith Runtime in the RS_RUN directory. These files require two to three megabytes of disk space. For more information about running reports, see*Creating Reports*.

Note For more information on deploying ReportSmith reports, refer to the file DEPLOY.TXT, installed to the DELPHI\DOC directory by default.

Appendix

A

Using Database Desktop

This appendix describes Database Desktop and provides a synopsis of Database Desktop features. The complete Database Desktop *User's Guide* is available on the Delphi CD-ROM, and all features are described in Database Desktop Help.

What is Database Desktop?

Database Desktop provides an easy way to create, restructure, and query tables to help you develop database applications with Delphi. You can use Database Desktop either as a standalone application on a single computer running Windows or as a multiuser application on a network.

This appendix contains information on using Database Desktop to work with data tables in a variety of formats.

The Database Desktop window

This section discusses the Database Desktop window and its menus.

Starting Database Desktop

Database Desktop

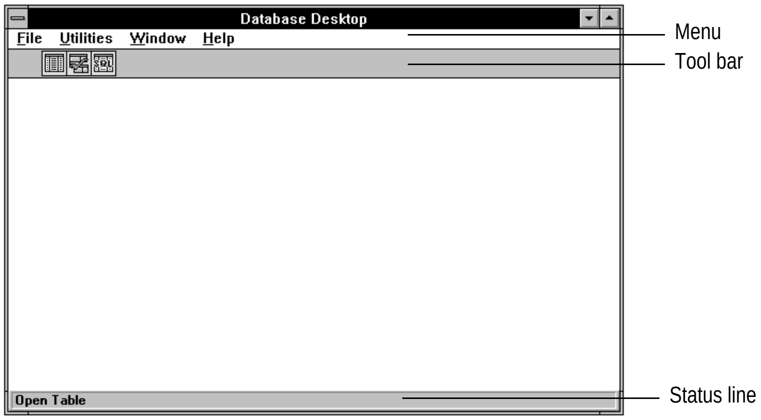
To start Database Desktop, double-click the Database Desktop icon in the Delphi program group or choose File | Run in the Program Manager and run DBD.EXE.

Database Desktop has several command-line options that let you control its configuration. For information on each option and its use, search for "command-line configuration" in the keyword list in Database Desktop Help.

The Database Desktop window

The first time you start Database Desktop, the Database Desktop window opens. All Database Desktop windows are opened in and contained by this window.

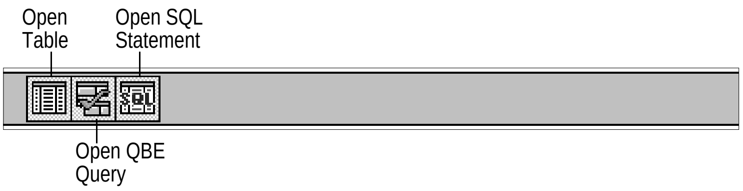
Figure A.1 The Database Desktop application window



Files you open in Database Desktop appear in their own type of windows. Tables appear in Table windows, queries appear in Query windows, and SQL statements appear in the SQL Editor.

Below the menu is a tool bar. The tool bar changes when the active window changes. The following figure shows the application window tool bar.

Figure A.2 Application window tool bar



Managing files

In Database Desktop you work with three types of files: QBE queries, .SQL files, and tables. Other files are created automatically by Database Desktop.

For a list of file extensions used by Database Desktop, search for “file-name extensions” in the keyword list in Database Desktop Help, and choose the topic “Types of Files.”

Opening files

To open a QBE query, SQL statement, or table, follow these steps:

1. Choose File | Open.
2. Choose the type of file to open—QBE query, SQL statement, or table.

3 Specify the file to open.

The Select File dialog box appears. For detailed information on the Select File dialog box, search for "Select File dialog box" in the keyword list in Database Desktop Help.

Note To access tables stored on a network, you must specify the location of the network control file. You do this by running the BDE Configuration Utility; double-click the BDE Configuration Utility icon in the Delphi program group. See online Help in the BDE Configuration Utility for more details.

Setting up a working directory

The *working directory* is where Database Desktop looks first for files. The Working Directory setting controls what files are listed in File | Open and File | Save dialog boxes. So, for example, if you want to open C:\DBD\SAMPLES\BOOKORD.DB, make C:\DBD\SAMPLES your working directory so that you see BOOKORD.DB when you choose File | Open | Table.

To specify a working directory, choose File | Working Directory, then type the path to the directory.

SQL You cannot set your working directory to an alias on a remote server.

Setting up a private directory

You should store temporary tables, such as *Answer*, in a private directory so they do not get overwritten by other users or applications. Choose File | Private Directory to establish a private directory.

Files stored in your private directory are listed in File | Open and File | Save dialog boxes, preceded by `:PRIV:` . Private directory files are visible and available to you, but not to other network users.

Aliases

You can assign an *alias* as a shorthand for a directory using the Alias Manager dialog box. For example, if you have a collection of tables and queries in one directory (called C:\DBD\PROJECTS\CUSTLIST), you can specify the alias :MYWORK: rather than type the entire path.

Using aliases, you can avoid typing long path names, and you can use the Path list in File | Open and File | Save dialog boxes to list files in any directory for which you have defined an alias.

To create an alias, choose File | Aliases. For information on creating, changing, or removing an alias, search for "aliases" in the keyword list in Database Desktop Help.

Creating tables

This section describes tables and discusses how to create and restructure Paradox, dBASE, and SQL tables in Database Desktop.

Understanding tables

A database is an organized collection of information or data. An address book is an example of a database. It organizes data about people into specific categories: names, phone numbers, and addresses.

In a relational database, the data is organized into tables. Each row of a table contains information about a particular item; this is called a *record*. Each column contains one piece of the information that makes up a record; this is called a *field*.

Figure A.3 A table

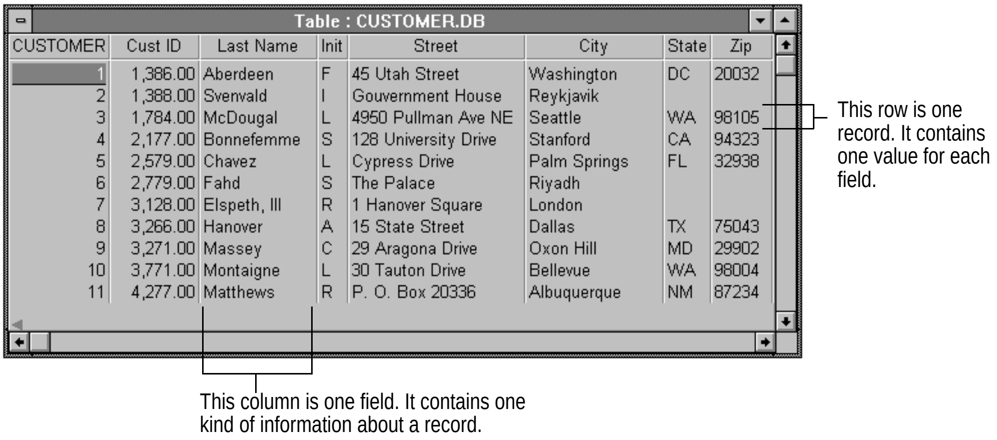


Table : CUSTOMER.DB
This row is one record. It contains one value for each field.
This column is one field. It contains one kind of information about a record.

CUSTOMER	Cust ID	Last Name	Init	Street	City	State	Zip
1	1,386.00	Aberdeen	F	45 Utah Street	Washington	DC	20032
2	1,388.00	Svenvald	I	Gouvernment House	Reykjavik		
3	1,784.00	McDougal	L	4950 Pullman Ave NE	Seattle	WA	98105
4	2,177.00	Bonnefemme	S	128 University Drive	Stanford	CA	94323
5	2,579.00	Chavez	L	Cypress Drive	Palm Springs	FL	32938
6	2,779.00	Fahd	S	The Palace	Riyadh		
7	3,128.00	Elspeth, III	R	1 Hanover Square	London		
8	3,266.00	Hanover	A	15 State Street	Dallas	TX	75043
9	3,271.00	Massey	C	29 Aragona Drive	Oxon Hill	MD	29902
10	3,771.00	Montaigne	L	30 Tauton Drive	Bellevue	WA	98004
11	4,277.00	Matthews	R	P. O. Box 20336	Albuquerque	NM	87234

Relational tables

Relational database applications such as dBASE and Paradox give you a way to link tables by comparing values stored in comparable fields in separate tables. The advantage of a relational database is that you can easily extract or combine data from several tables to get exactly the information you need, without changing the structure of the database. Also, a few small and discrete tables are more convenient to use and maintain than one large table.

The sample database files CUSTOMER.DB and BOOKORD.DB are examples of relational tables. These tables can be linked through the fields containing customers' ID numbers (*Cust ID* in the *Customer* table, Cust in *Bookord*). When the tables are linked, you can extract information from both tables into one table. For example, you can search for and extract a list of quantities ordered (from the *Bookord* table) and the respective last names (from *Customer*). The results are returned in an *Answer* table (see page 170).

Planning tables

Planning is the first step in creating a table. Decide what you want the table to contain and how you want to lay it out. For tips on planning tables, search for "planning" in the keyword list in Database Desktop Help, and choose the topic "Planning Tables."

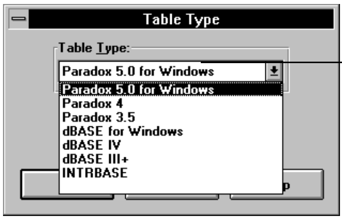
Creating a new table

To create a new table,

1 Choose File | New | Table. Or right-click the Open Table tool bar button, and choose New.

The Table Type dialog box appears.

Figure A.4 Table Type dialog box



Choose the type of table you want to create. Some options discussed in this appendix are available only to Paradox for Windows 5.0 tables.

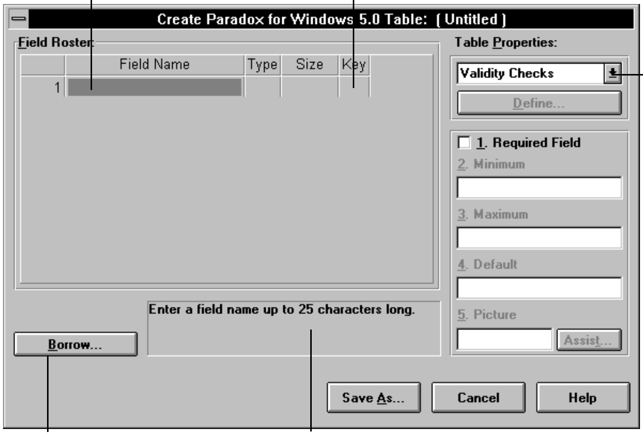
2 If you want a table type other than Paradox for Windows, click the arrow next to the list box and select from the drop-down list.

3 Choose OK. The Create Table dialog box appears. This dialog box may have a slightly different appearance for different table types, but it will function the same.

Figure A.5 The Create Table dialog box

Enter the field name, type, and size in the Field Roster.

Press any key or double-click to key the table.



Enter a field name up to 25 characters long.

When the dialog box is opened, the Validity Checks table property is selected and all types of validity checks are available.

Choose this to borrow the structure of another table.

The status box gives you guidelines as you create the table.

For a step-by-step description of creating a table, search for "creating tables" in the keyword list in Database Desktop Help, and choose the topic "Creating a New Table."

Defining fields

Use the Field Roster in the Create Table dialog box (page 151) to define the fields of the new table. You can use the mouse, arrow keys, *Enter*, *Tab*, or *Shift+Tab* to move among the columns. (*Shift+Tab* moves backwards.)

Field names

Type field names in the Field Name column of the Field Roster.

For information on rules governing field names, search for "field names" in the keyword list in Database Desktop Help, and choose the topic for the type of table you are using (Paradox, dBASE, or SQL).

Adding, deleting, and rearranging fields

You can add, delete, and rearrange fields in the Field Roster.

For detailed information on changing fields, search for "fields" in the keyword list in Database Desktop Help."

Specifying field type

To specify the field type in the Create Table dialog box,

1. Select the Type column of the field you want.
2. Type the symbol (or name, for SQL tables) for the field type or select from the drop-down list. You can use the list in two ways:
   - Right-click the Type column again and click to select the field type.
   - Press *Spacebar* to see the list, then choose the field type.

For information on field types and sizes, search for "field types" in the keyword list in Database Desktop Help, and choose the topic for the type of table you are using (Paradox, dBASE, or SQL).

Using indexes

The BDE uses indexes to keep track of the location of records in tables. This makes it easy to maintain a sorted order of a table and view like values together.

When you create an index for a Paradox or dBASE table, Database Desktop creates a file that contains the indexed field's values and their locations. Database Desktop uses the index file to locate and sort the records in a table.

Indexes work differently for Paradox, dBASE, and SQL tables.

Keys in Paradox tables

In Paradox tables, the primary index is called the *key*. A Paradox table's key establishes the primary index and sort order for the table. A key also requires each value in the field(s) that defines the key to be unique. For example, if the Cust ID field is identified as the key of the *Customer table*, each value in the Cust ID field must be unique. Likewise, if the Cust, Date, and Item # fields are identified as the key of the *Bookord table*, the field values (taken as an ordered group) must be unique. This guards against duplication of data within the table. Keys are required for linking tables and for using the data integrity features of Paradox tables.

To create a key, display the Create Table (page 151) or the Restructure Table (page 160) dialog box. Then move to the Key column in the Field Roster and double-click (or press any key). The key field indicator (*) appears. Database Desktop keys the table on the selected field.

Follow these rules when defining keys:

- A table can have only one primary key. This key can be made up of one or more fields.
- If a key is defined as a single field, that field must be the first field in the Field Roster.
- If you identify more than one field as keyed, you create a *composite* key. These fields, taken as a group, must be unique for each record of the table. The composite key must be the first fields in the Field Roster.

A dBASE table's index

When working with dBASE tables, Database Desktop uses an index to organize the records in a table according to the values in one or more fields.

When you create an index on a dBASE table, a file is created that contains the indexed field's values and their corresponding record numbers. Database Desktop refers to the index file when locating and displaying the records in a table.

Although Database Desktop supports both .MDX files and .NDX files, it is recommended that you use a dBASE production index (the .MDX file which uses the table's name as its file name) whenever possible. Although you can create nonproduction .MDX files as well as .NDX files, Database Desktop automatically maintains the production index.

For more information on dBASE indexes, search for "dBASE indexes" in the keyword list in Database Desktop Help.

Note Delphi does not support all dBASE index types. For more information, see Chapter 3, "Using data access components and tools."

An SQL table's index

SQL tables use unique and non-unique indexes, but they do not use the primary keys that Paradox tables use. You can create multiple indexes for an SQL table; for each index, you specify whether it is unique or non-unique. Depending on the server, you may also be able to specify whether the index is case-sensitive and whether it is ascending or descending. SQL indexes, unlike Paradox and dBASE indexes, are always maintained.

You can use Database Desktop to create and modify indexes on SQL tables, but you cannot specify which index to use in Database Desktop.

When you use an SQL table in Database Desktop, the table should have a unique index. If it does not have a unique index you may not be able to view new records that you insert, depending on the server. To add a unique index to an existing table, choose Utilities | Restructure.

For more information on SQL indexes, see "Creating indexes on SQL tables" on page 155.

Defining secondary indexes

A *secondary index* is a field or group of fields other than the key field that can be used to sort the table or to link the table to other tables. Database Desktop enables you to create secondary indexes for Paradox tables.

You can use a secondary index to see an alternate view order for a Paradox table. For example, to view the *Customer* table by City, while keeping the table's key order intact, you can use a secondary index on City to temporarily change the view order of the records.

For information on creating a secondary index, search for "secondary indexes" in the keyword list in Database Desktop Help, and choose the topic "Defining a Secondary Index."

Specifying validity checks

Validity checks govern the values you can enter in a field. The five types of validity checks are listed in Table A.1.

Note Validity checks work only on Paradox tables, not on dBASE tables. For SQL tables, the only validity check you can specify in Database Desktop is whether a field is required (not Null).

Table A.1 Paradox validity checks

Validity check	Meaning
Required field	Every record in the table must have a value in this field. SQL tables can also use this validity check (equivalent to NOT NULL).
Minimum	The values entered in this field must be equal to or greater than the minimum you specify here.
Maximum	The values entered in this field must be less than or equal to the maximum you specify here.
Default	The value you specify here is automatically entered in this field. You can replace it with another value.
Picture	You specify a character string that acts as a template for the values that can be entered in this field. The values entered in this field are automatically formatted according to this picture. For information on pictures, search for "picture strings" in the keyword list in Database Desktop Help, and choose the topic "Picture String Characters."

For detailed information on validity checks, search for "validity checks" in the keyword list in Database Desktop Help.

Note A "Required field" validity check will set a Delphi *TField*'s *Required* property to True when the table is accessed from a Delphi application.

Borrowing a table structure

When creating a table similar to one you already have, you can borrow its structure. Then you can either use it as is or change it. You must begin from a blank table structure to borrow another table's structure.

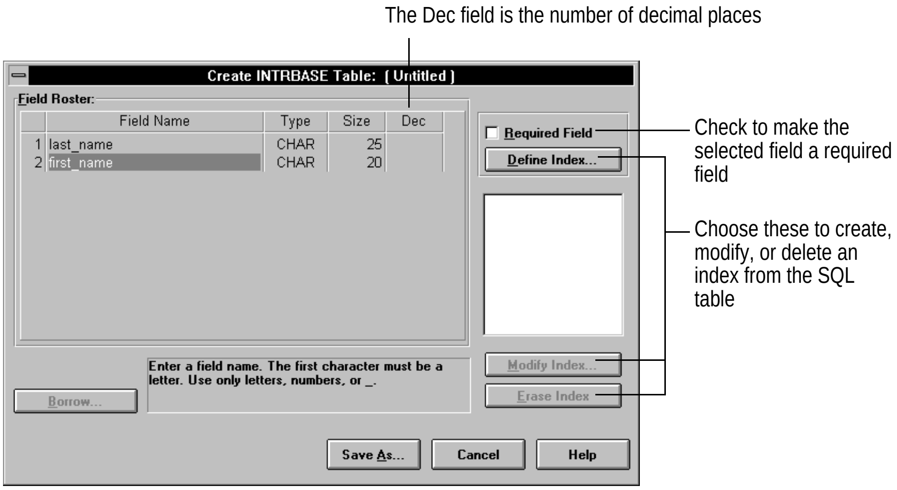
To borrow a table structure, choose Borrow from the Create Table dialog box.

For detailed information on borrowing, search for "borrow a table's structure" in the keyword list in Database Desktop Help.

Creating an SQL table

When you create an SQL table, you can define the table structure (fields & types), specify required fields, and define indexes. The Create Table dialog box for SQL tables looks as shown in Figure A.6.

Figure A.6 The Create Table dialog box for SQL tables


The Dec field is the number of decimal places
Create INTRBASE Table: [Untitled]
Field Roster:
Required Field
Define Index...
Modify Index...
Erase Index
Borrow...
Save As...
Cancel
Help
Check to make the selected field a required field
Choose these to create, modify, or delete an index from the SQL table

	Field Name	Type	Size	Dec
1	last_name	CHAR	25	
2	first_name	CHAR	20	

Enter a field name. The first character must be a letter. Use only letters, numbers, or _.

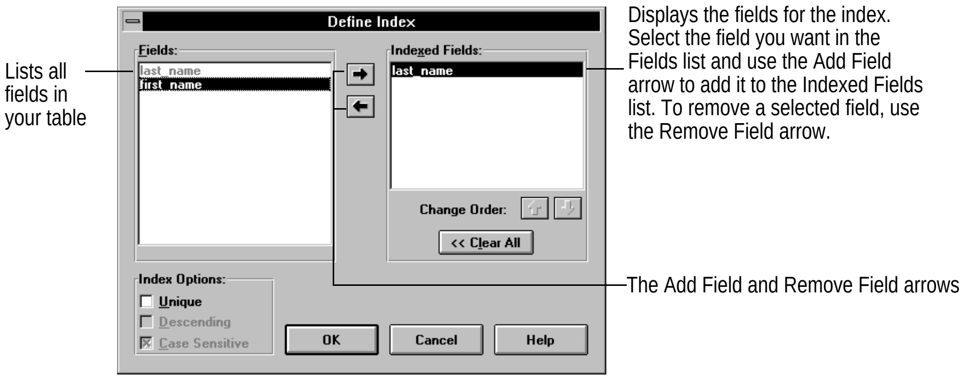
For information on valid field types for your SQL server, search for "field types" in the keyword list in Database Desktop Help, and choose the topic for your server.

Creating indexes on SQL tables

You can use Database Desktop to create and modify indexes on SQL tables.

To create an index for an SQL table, display the Create Table (page 155) or the Restructure Table (page 160) dialog box. Then, choose Define Index. Database Desktop displays the Define Index dialog box, shown in Figure A.7.

Figure A.7 The Define Index dialog box for SQL indexes


Define Index
Fields:
last_name
first_name
Indexed Fields:
last_name
Change Order:
<< Clear All
Index Options:
Unique
Descending
Case Sensitive
OK
Cancel
Help
Lists all fields in your table
Displays the fields for the index. Select the field you want in the Fields list and use the Add Field arrow to add it to the Indexed Fields list. To remove a selected field, use the Remove Field arrow.
The Add Field and Remove Field arrows

When you use an SQL table in Database Desktop, the table should have a unique index. If it does not have a unique index, you may not be able to view new records you insert.

For detailed information on using the Define Index dialog box, search for "Define Index Dialog Box" in the keyword list in Database Desktop Help, and choose the topic "Define Index Dialog Box (SQL Tables)."

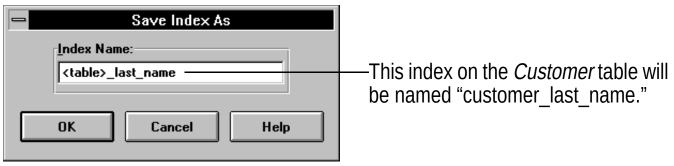
Naming SQL indexes

For most database servers, index names must be unique within the database (or in some other predefined workspace). When you create an index on an SQL table, Database Desktop prompts you to prefix the index name with the table name to ensure that the index name is unique.

Sybase note Sybase index names need only be unique within a table, not within the entire database, so Database Desktop does not prefix Sybase index names with table names.

When you create an SQL index and choose OK from the Define Index dialog box, Database Desktop supplies the prefix "<table>_" for the index name as follows:

Figure A.8 Save Index As dialog box


Save Index As
Index Name:
<table>_last_name
OK
Cancel
Help
This index on the *Customer* table will be named "customer_last_name."

You can include the table name with the index name or omit it:

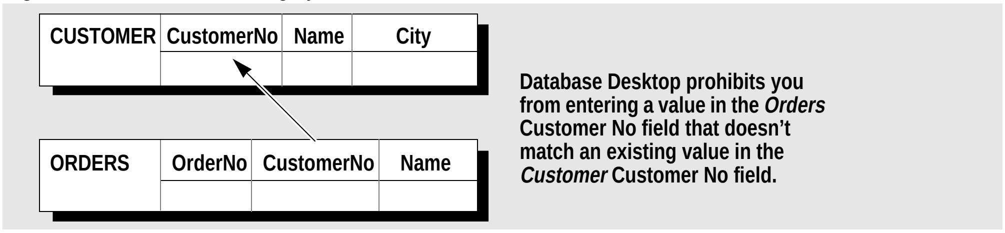
- If you type the index name following "<table>_", Database Desktop prefixes the index name with the table name and an underscore.
- If you delete "<table>_", Database Desktop omits the table name from the index name. If the index name is not unique, an error will occur when Database Desktop saves the table.

This index naming scheme also affects restructuring , as described in "Restructuring an SQL table" on page 160.

Defining referential integrity for Paradox tables

Referential integrity means that a field or group of fields in one table (the "child" table) must refer to the key of another table (the "parent" table). Database Desktop enables you to define referential integrity rules for Paradox tables.

Figure A.9 Referential integrity


CUSTOMER
CustomerNo
Name
City
ORDERS
OrderNo
CustomerNo
Name
Database Desktop prohibits you from entering a value in the *Orders* Customer No field that doesn't match an existing value in the *Customer* Customer No field.

Database Desktop accepts only those values that exist in the parent table's key as valid values for the specified field(s) of the child table. You can establish referential integrity only between like fields that contain matching values. For example, you can establish referential integrity between *Customer* and *Orders* on their CustomerNo fields. In both cases, the values contained in the specified fields are the same. The field names don't matter as long as the field types and sizes are identical.

Note You can establish referential integrity only between tables in the same directory.

Using referential integrity, Database Desktop checks the validity of a value before accepting it in the table. If you establish referential integrity between *Customer* and *Orders* on their CustomerNo fields, then enter a value in the CustomerNo field of *Orders*, Database Desktop searches the CustomerNo field of *Customer* and

- Accepts the value in *Orders* if it exists in *Customer*
- Rejects the value in *Orders* if it doesn't exist in *Customer*

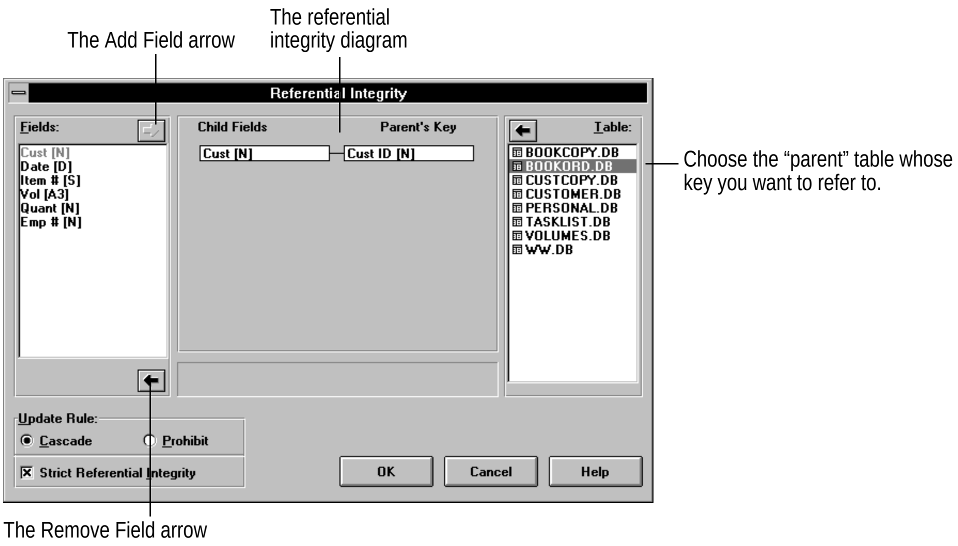
Procedure

To define a referential integrity relationship,

1 In the Create Table (page 151) or Restructure Table (page 160) dialog box, choose Referential Integrity from the Table Properties list. The Define button becomes available.

2 Choose Define to open the Referential Integrity dialog box.

Figure A.10 Referential Integrity dialog box



Fields: Cust [N], Date [D], Item # [S], Vol [A3], Quant [N], Emp # [N]

Child Fields: Cust [N] — Parent's Key: Cust ID [N]

Table: BOOKCOPY.DB, BOOKORD.DB, CUSTCOPY.DB, CUSTOMER.DB, PERSONAL.DB, TASKLIST.DB, VOLUMES.DB, WW.DB

Update Rule: (●) Cascade ( ) Prohibit

[✓] Strict Referential Integrity

3 Choose the parent table from the Table list. The table’s key field appears in the Parent’s Key area of the referential integrity diagram.

4 Double-click the child table’s field in the Fields list (or *Tab* to it and click the Add Field arrow or press *Alt+A*). The field name appears in the Child Fields area of the referential integrity diagram.

5 Choose the update rule you want.

6 Choose whether you want to enforce strict referential integrity.

7 Choose OK to name and save the referential integrity relationship.

For detailed information on defining referential integrity, search for “Referential Integrity” or “Referential Integrity Dialog Box” in the keyword list in Database Desktop Help.

Creating table lookup

Table lookup helps you enter data in one Paradox table that already exists in the first field of another Paradox table—the *lookup* table. Table lookup lets you

- Require that the values you enter into a field exist in the first field of another table
- Refer to another table to look up the acceptable values for a field
- Copy values in the lookup table to the table you’re editing

Table lookup is primarily a data entry tool. Unlike referential integrity, it doesn’t track or control changes you make to the lookup table. Table lookup ensures that data is copied accurately from one table to another; referential integrity ensures that the ties between data in separate tables cannot be broken.

For detailed information on table lookup, search for “table lookup” in the keyword list in Database Desktop Help, and choose the topic “Looking up Table Values.”

Note Table lookup rules have no effect on Delphi applications accessing a table.

Establishing passwords for Paradox tables

Sometimes it’s important to ensure that the Paradox table you create is protected from access by unauthorized users. Not only can you establish a password for a Paradox table as a whole, but you can also assign specific rights to the table or individual fields.

For detailed information on creating and using passwords, search for “passwords” in the keyword list in Database Desktop Help.

Restructuring tables

Database Desktop enables you to restructure Paradox and dBASE tables to:

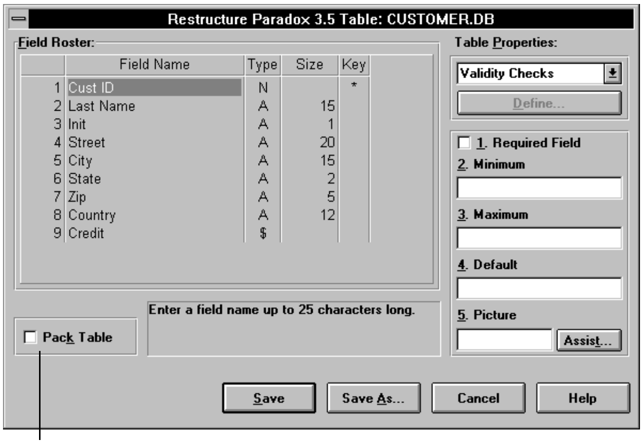
- Add or rename fields
- Change field types or sizes
- Modify indexes
- Modify table language drivers

Restructuring a table is very much like creating it for the first time. You will not be able to restructure a table if Delphi or any other application has the table open. For detailed information on restructuring tables, see the Database Desktop Help; choose Help | Contents | Tasks | Creating and Restructuring | Restructuring a Table.

Note Restructuring a table may require corresponding modifications in Delphi applications that access the table. For example, removing a column or changing its data type will raise an exception if an application has a persistent *TField* component for the modified column.

To restructure a table, choose Utilities | Restructure, then choose the table you want. If the table you want to restructure is already open in the active window, use Table | Restructure. The Restructure Table dialog box opens. The following figure shows an example of a Paradox table in the Restructure Table dialog box:

Figure A.11 The Restructure Table dialog box for Paradox tables



Restructure Paradox 3.5 Table: CUSTOMER.DB

Field Roster:

	Field Name	Type	Size	Key
1	Cust ID	N		*
2	Last Name	A	15	
3	Init	A	1	
4	Street	A	20	
5	City	A	15	
6	State	A	2	
7	Zip	A	5	
8	Country	A	12	
9	Credit	$		

Table Properties: Validity Checks

Enter a field name up to 25 characters long.

Work in the Restructure Table dialog box the same way you work in the Create Table dialog box.

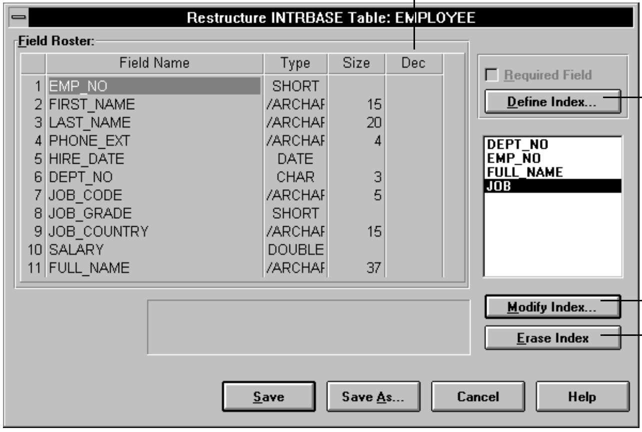
Pack a table to reuse disk space left over from deleting records. Some restructure operations automatically pack your table. You can check Pack Table and choose OK when you want to be sure Paradox packs the table.

Restructuring an SQL table

When you restructure an SQL table, you can add, modify, and drop indexes. You cannot otherwise change the structure of a table on a server. The Restructure Table dialog box for SQL tables looks as shown in Figure A.12.

Figure A.12 The Restructure Table dialog box for SQL tables

The Dec field is the number of decimal places



Restructure INTRBASE Table: EMPLOYEE

Field Roster:

	Field Name	Type	Size	Dec
1	EMP_NO	SHORT		
2	FIRST_NAME	/ARCHAF	15	
3	LAST_NAME	/ARCHAF	20	
4	PHONE_EXT	/ARCHAF	4	
5	HIRE_DATE	DATE		
6	DEPT_NO	CHAR	3	
7	JOB_CODE	/ARCHAF	5	
8	JOB_GRADE	SHORT		
9	JOB_COUNTRY	/ARCHAF	15	
10	SALARY	DOUBLE		
11	FULL_NAME	/ARCHAF	37	

DEPT_NO
EMP_NO
FULL_NAME
JOB

Choose these to create, modify, or delete an index from the SQL table

Prefixing the index name with the table name

Database Desktop prefixes some index names with the table name, as described in “Creating indexes on SQL tables” on page 155. These index names are also affected when you restructure an SQL table as follows:

- If you create a new index during a restructure, Database Desktop prompts you to prefix the index name with the table name.
- If you modify an index during a restructure, Database Desktop does not modify the index name, unless you rename the index as part of your modification.
- If you choose Save As during a restructure, Database Desktop prefixes all index names with the new table name, if you have not explicitly entered an index name. For example, suppose the EMPLOYEE table contains the following indexes:

  EMPLOYEE_DEPT_NO
  EMPLOYEE_EMP_NO
  FULL_NAME
  JOB

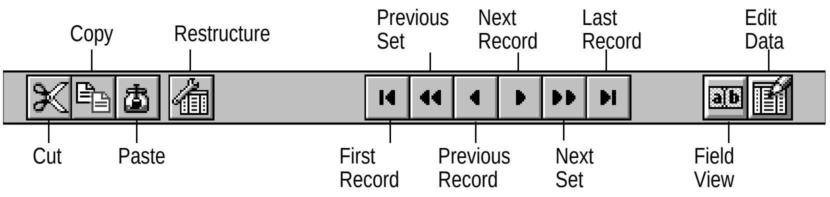
  If you restructure the table and save it as MY_DEPT, Database Desktop renames the indexes as follows:

  MY_DEPT_DEPT_NO
  MY_DEPT_EMP_NO
  MY_DEPT_FULL_NAME
  MY_DEPT_JOB

Viewing tables

To open a table, choose File | Open | Table. Or, if the application window is empty, click the Open Table button in the tool bar. In the Open Table dialog box, choose the table to open. The table you chose opens in a Table window, and the tool bar appears as shown in the following figure:

Figure A.13 The Table window tool bar



When you first open a table, its data appears in a Table window in View mode. Each Table window contains an independent view of a table, so different views of a single table can be open at the same time. Up to 24 tables can be open at one time.

To be able to simultaneously access tables stored on a network, you must provide Database Desktop the location of the *network control file*. You do this by running the BDE Configuration Utility; double-click the BDE Configuration Utility icon in the Delphi program group. See online Help in the BDE Configuration Utility for more details.

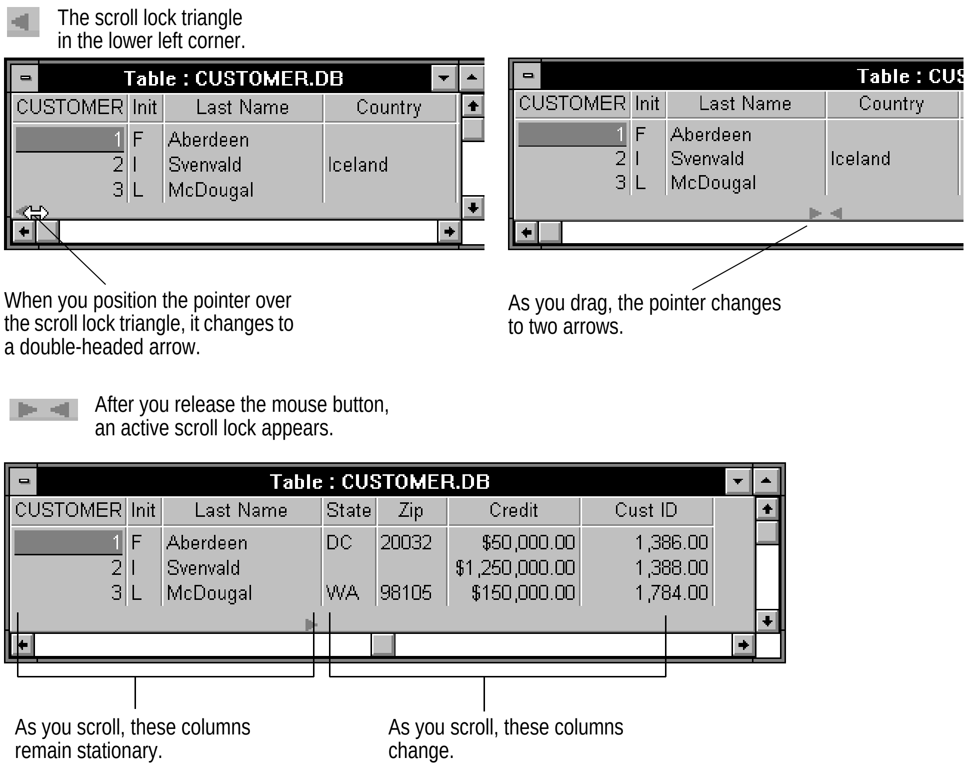
A Delphi application can specify the network control file in the *NetFileDir* property of *TSession*. This enables a Delphi application access to tables stored on a network.

Using scroll lock

To lock one or more columns in place as you move horizontally through the table's columns, use a *scroll lock*. All columns to the left of the lock remain stationary as you move through the table's columns.

The scroll lock is a triangle in the lower left corner of the Table window. To place a lock, drag the triangle to the right side of the column(s) you want to lock. An active scroll lock appears as two triangles when you release the mouse button, as shown in Figure A.14.

Figure A.14 A scroll lock in the Table window


The scroll lock triangle in the lower left corner.
Table : CUSTOMER.DB
When you position the pointer over the scroll lock triangle, it changes to a double-headed arrow.
Table : CUS
As you drag, the pointer changes to two arrows.
After you release the mouse button, an active scroll lock appears.
Table : CUSTOMER.DB
As you scroll, these columns remain stationary.
As you scroll, these columns change.

CUSTOMER	Init	Last Name	Country
1	F	Aberdeen	
2	I	Svenvald	Iceland
3	L	McDougal	

CUSTOMER	Init	Last Name	State	Zip	Credit	Cust ID
1	F	Aberdeen	DC	20032	$50,000.00	1,386.00
2	I	Svenvald			$1,250,000.00	1,388.00
3	L	McDougal	WA	98105	$150,000.00	1,784.00

Customizing a table view

The *view* of a table is how it appears onscreen; you can modify and save a custom view of a table. Changing the view makes it easier to see specific fields; the actual *structure* of the table (its definition of field order and size) remains the same. To customize a view, you can rearrange, resize, and lock columns, and resize rows or table headings.

The following figure shows the *hot zones* on an open table view. Hot zones indicate areas on a table where you can drag to modify the view of the table. As the pointer passes over a hot zone, the pointer changes shape.

Figure A.15 Hot zone pointers in the Table window

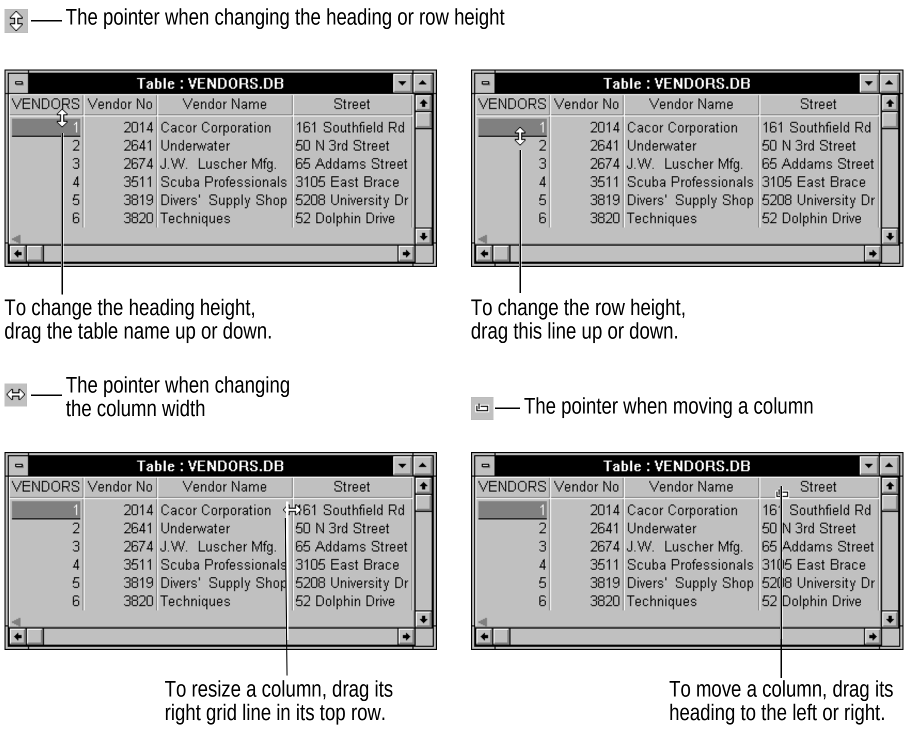

The pointer when changing the heading or row height
To change the heading height, drag the table name up or down.
To change the row height, drag this line up or down.
The pointer when changing the column width
The pointer when moving a column
To resize a column, drag its right grid line in its top row.
To move a column, drag its heading to the left or right.

Table : VENDORS.DB

VENDORS	Vendor No	Vendor Name	Street
1	2014	Cacor Corporation	161 Southfield Rd
2	2641	Underwater	50 N 3rd Street
3	2674	J.W. Luscher Mfg.	65 Addams Street
4	3511	Scuba Professionals	3105 East Brace
5	3819	Divers' Supply Shop	5208 University Dr
6	3820	Techniques	52 Dolphin Drive

Rearranging and resizing columns

To move a column, position the pointer on a column's heading. When the pointer changes shape (shown at left), drag the column to its new position.

To resize a column, position the pointer on its right boundary line (either the heading area or the top row of data). When the pointer changes shape (shown at left), drag the boundary line to increase or decrease the width of the column.

Resizing rows

To resize the height of all of the rows in a table, drag the line under the first record number. Move the line up to decrease the row height, or down to increase the row height.

Resizing column headings

To resize the height of all the column headings, drag the line under the table name. The table name is located above the left-most column (which contains the record numbers).

Saving a custom view

You can save a custom view, undo changes to a view, and restore the default view by using the commands on the Properties menu.

For detailed information on the Properties menu commands, search for "Properties menu" in the keyword list in Database Desktop Help.

Editing data

This section introduces Database Desktop's editing features. For detailed information on editing data, search for "editing data" or "Edit mode" in the keyword list in Database Desktop Help.

Using Edit mode

To change data in a table, you must be in Edit mode. To enter Edit mode, do one of the following:

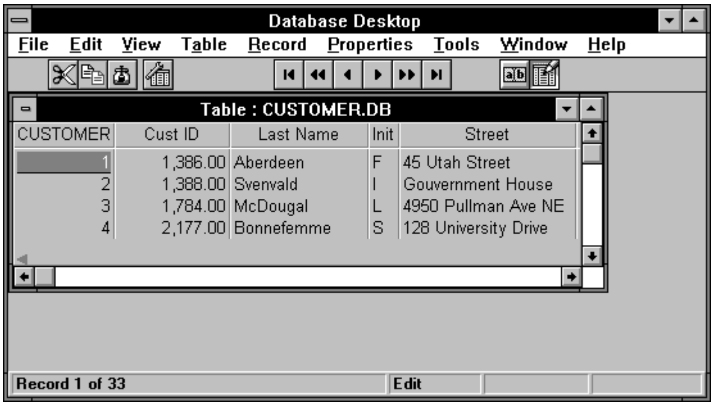
- Click the Edit Data button in the tool bar.
- Choose View | Edit Data.
- F9 Press *F9*.

In Edit mode, you can select any field and begin typing to replace its existing entry. When you enter Edit mode, the Edit Data button remains pressed in and the status line tells you Edit mode is active.

Note In Database Desktop you cannot edit data in the following field types:

- **Paradox:** Memo, Formatted Memo, Graphic, OLE, Autoincrement, Binary, or Bytes
- **dBASE:** Memo, OLE, or Binary
- **SQL:** any BLOB (binary large object) field or a text field that allows more than 255 characters

Figure A.16 The *Customer* table in Edit mode



CUSTOMER	Cust ID	Last Name	Init	Street
1	1,386.00	Aberdeen	F	45 Utah Street
2	1,388.00	Svenvald	I	Gouvernment House
3	1,784.00	McDougal	L	4950 Pullman Ave NE
4	2,177.00	Bonnefemme	S	128 University Drive

Record 1 of 33 Edit

Selecting fields and records

When you move to a field or click it, the field is highlighted. This indicates that the field is *selected*. In Edit mode, if you type anything into a selected field, you'll replace the existing entry with the value you type. The cut, copy, and paste operations affect the entire field entry when it's selected.

You can select more than one field at a time, or select a portion of a single field entry.

For detailed information on selecting fields and records, search for "selecting data" in the keyword list in Database Desktop Help.

Field view

In Edit mode, you can change a field's entry in one of two ways:

- Select the field and type a new value. When you begin typing, the new value replaces the old entry.
- Select the field and edit the existing entry using *field view*.

For information on field view, search for "Field View" in the keyword list in Database Desktop Help.

Adding, subtracting, and emptying records

Use the commands on the Utilities menu to add, subtract, or empty a table's records.

- **Add:** You can add the records from one table to another table. You can use Add on all table types. Choose Utilities | Add. Database Desktop opens the Add dialog box.

  For detailed information on adding records, see Database Desktop Help; choose Help | Contents | Tasks | Using Table Utilities | Adding Records.
- **Subtract:** You can remove records that exist in one table from a different table by using the Subtract utility. You can subtract records only from a keyed Paradox table. Choose Utilities | Subtract. Database Desktop opens the Subtract dialog box.

  For detailed information on subtracting records, see Database Desktop Help; choose Help | Contents | Tasks | Using Table Utilities | Subtracting Records.

  SQL You cannot use an SQL table as the source of a Subtract operation.
- **Empty:** Use the Empty utility to remove all records from a table, leaving the table's structure (including all keys, indexes, validity checks, and so on) intact. You can use Empty on Paradox, dBASE, and SQL tables. Choose Utilities | Empty. Database Desktop opens the Empty dialog box.

  For detailed information on emptying tables, see Database Desktop Help; choose Help | Contents | Tasks | Using Table Utilities | Emptying Tables.

Sorting, copying, renaming, and deleting objects

You can use Database Desktop to sort tables and to copy, rename, and delete objects.

Sorting tables

When you sort a table, you tell Database Desktop to rearrange the order of the records in the table and display them in the order you specify.

- If a table is keyed, Database Desktop creates a new, unkeyed table containing the sorted data. The original table remains unchanged.
- If a table is not keyed, the sort changes the actual location of the records in the table.

SQL You cannot sort SQL tables.

To sort a table, choose Utilities | Sort, then choose the table you want to sort from the Select File dialog box. Database Desktop opens the Sort Table dialog box.

For detailed information on sorting tables, see Database Desktop Help; choose Help | Contents | Tasks | Using Table Utilities | Sorting Tables.

Copying objects

You can copy Paradox and dBASE tables, queries, SQL tables, and .SQL files from within Database Desktop. To copy an object, choose Utilities | Copy. Database Desktop opens the Copy dialog box.

When you copy a table, Database Desktop copies both the structure of the table and the data contained in it. You can copy tables from one table type to another. For example, you can copy a Paradox table to a dBASE or SQL table. To copy to an SQL table type, you must have an SQL database server and the appropriate SQL Link.

For detailed information on copying objects, see Database Desktop Help; choose Help | Contents | Tasks | Using Table Utilities | Copying Tables.

Renaming objects

You can rename tables, queries, and .SQL files from within Database Desktop. You cannot rename SQL tables.

To rename an object, choose Utilities | Rename. Database Desktop opens the Rename dialog box. For detailed information on renaming objects, see Database Desktop Help; choose Help | Contents | Tasks | Using Table Utilities | Renaming Tables.

Deleting objects

You can delete tables, queries, SQL tables, and .SQL files from within Database Desktop.

To delete an object, choose Utilities | Delete. Database Desktop opens the Delete dialog box. For detailed information on deleting objects, see Database Desktop Help; choose Help | Contents | Tasks | Using Table Utilities | Deleting Tables.

Executing SQL statements

This section describes how to use the SQL Editor to enter and execute SQL statements.

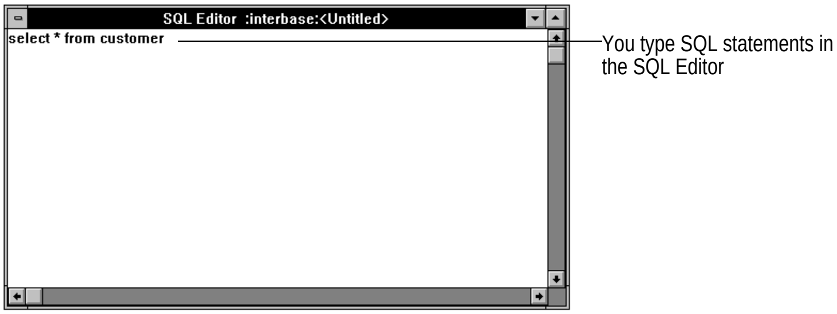
What is the SQL Editor?

Programmers familiar with SQL can use the SQL Editor window to directly enter, execute, or save an SQL statement. You can save the SQL statement to a file, and then later load, modify, or execute it.

In the SQL Editor, you can enter SQL statements to be executed by a SQL database server. The syntax must conform to your server's dialect. This is referred to as *passthrough SQL*. The SQL server performs all error or syntax checking and executes the statement without any involvement by Database Desktop.

You can also use the SQL Editor to run SQL statements against Paradox or dBASE tables. This is known as local SQL, and the allowable syntax is a subset of ANSI standard SQL. For more information, see Appendix C, "Using local SQL."

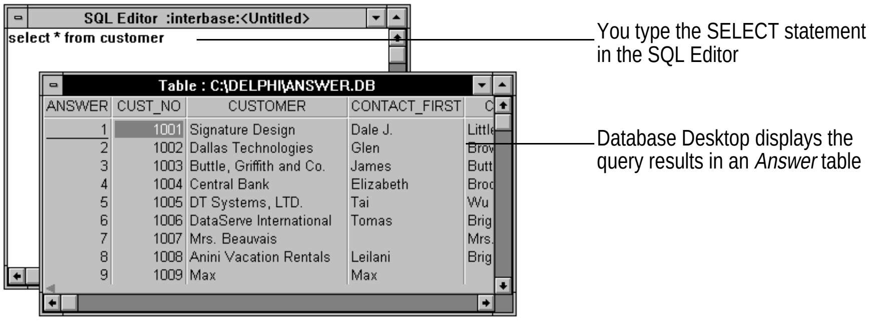
Figure A.17 The SQL Editor


SQL Editor :interbase:<Untitled>
You type SQL statements in the SQL Editor

```
select * from customer
```

If you execute a SELECT statement in the SQL Editor, Database Desktop displays the resulting data in an *Answer* table, as shown in Figure A.18.

Figure A.18 The SQL Editor and an *Answer* table

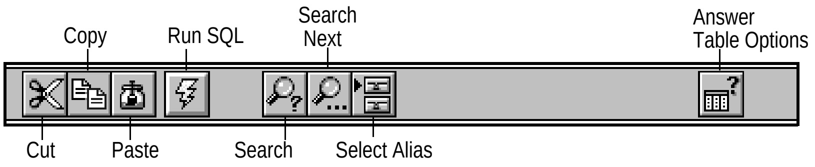

SQL Editor :interbase:<Untitled>
Table : C:\DELPHI\ANSWER.DB
You type the SELECT statement in the SQL Editor
Database Desktop displays the query results in an *Answer* table

```
select * from customer
```

ANSWER	CUST_NO	CUSTOMER	CONTACT_FIRST	C
1	1001	Signature Design	Dale J.	Little
2	1002	Dallas Technologies	Glen	Brow
3	1003	Buttle, Griffith and Co.	James	Butt
4	1004	Central Bank	Elizabeth	Broc
5	1005	DT Systems, LTD.	Tai	Wu
6	1006	DataServe International	Tomas	Brig
7	1007	Mrs. Beauvais		Mrs.
8	1008	Anini Vacation Rentals	Leilani	Brig
9	1009	Max	Max	

The SQL Editor has the tool bar shown in Figure A.19.

Figure A.19 SQL Editor Toolbar



Opening the SQL Editor

To open the SQL Editor, do one of the following:

To do this	Do this
Enter (and execute) a new SQL statement	Choose File | New | SQL Statement Or right-click the Open SQL Script tool bar button and choose New
Open (and edit or execute) an existing .SQL file	Choose File | Open | SQL Statement Or click the Open SQL Script tool bar button Or right-click the Open SQL Script tool bar button and choose Open
View the SQL equivalent of an open QBE query	Choose Query | Show SQL Or click the Open SQL Script tool bar button

Specifying an alias

Before running an SQL statement, you must specify the alias that the statement will run against. To specify an alias, do one of the following:

- Choose SQL | Select Alias.
- Click the Select Alias tool bar button.

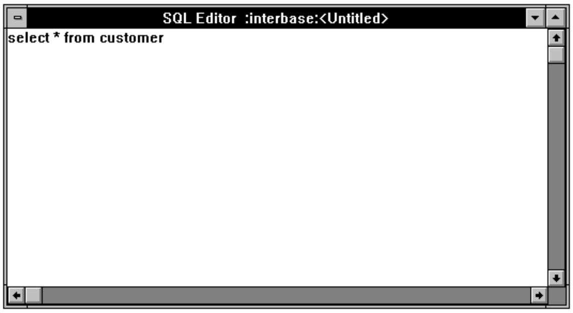
Database Desktop opens the Select Alias dialog box, where you can choose one of the aliases you created in the Alias Manager dialog box. You cannot include an alias in the text of the SQL statement.

Running an SQL statement

You can enter multiple SQL statements if your server allows it and you include only one SELECT statement.

To run an SQL statement that you have typed in the SQL Editor window, click the Run SQL tool bar button or choose SQL | Run SQL.

Figure A.20 SQL statement in the SQL Editor



```
select * from customer
```

If your SQL statement is a query, the query results are displayed in an *Answer* table, as shown in Figure A.18 on page 168.

Saving an SQL statement

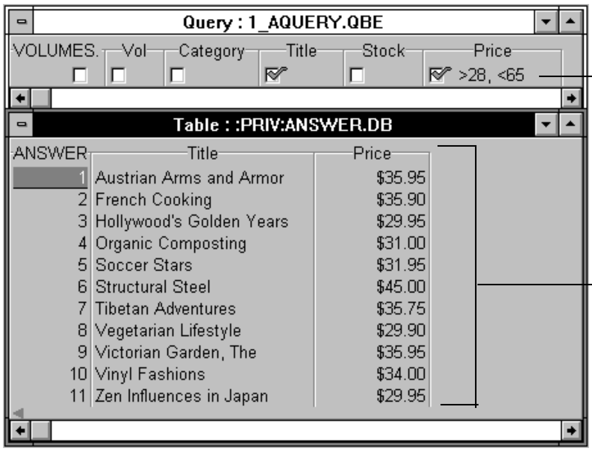
To save the SQL statement in the active window, choose File | Save or File | Save As. When you save an SQL statement to your local hard disk, Database Desktop places it in an unformatted text file with an .SQL extension.

Querying table data with QBE

A *query* is a question you ask about information in one or more tables. In addition to standard SQL queries, Database Desktop enables you to use a technique called query by example (QBE) to extract and manipulate data in tables. With QBE, you make the query image look like an example of the records you want to search for.

The following figure shows a query that gives examples of the fields you want to see (and a range of values within one of those fields), and the answer Database Desktop gives.

Figure A.21 A query and its results



VOLUMES.	Vol	Category	Title	Stock	Price
☐	☐	☐	✓	☐	✓ >28, <65

ANSWER	Title	Price
1	Austrian Arms and Armor	$35.95
2	French Cooking	$35.90
3	Hollywood's Golden Years	$29.95
4	Organic Composting	$31.00
5	Soccer Stars	$31.95
6	Structural Steel	$45.00
7	Tibetan Adventures	$35.75
8	Vegetarian Lifestyle	$29.90
9	Victorian Garden, The	$35.95
10	Vinyl Fashions	$34.00
11	Zen Influences in Japan	$29.95

This example searches for book prices greater than $28 and less than $65 (the checkmarks specify which fields appear in the *Answer* table).

The *Answer* table displays the checked fields for records that match the example.

The result of a query is a temporary table called *Answer*. The *Answer* table is overwritten each time a query is run.

SQL If you have Borland SQL Links, you can use QBE to view and query tables on SQL servers. For more information, see the Database Desktop Help Contents.

For details on using QBE, see Database Desktop Help; choose Help | Contents | Tasks | Using Query-By-Example.

Appendix

B

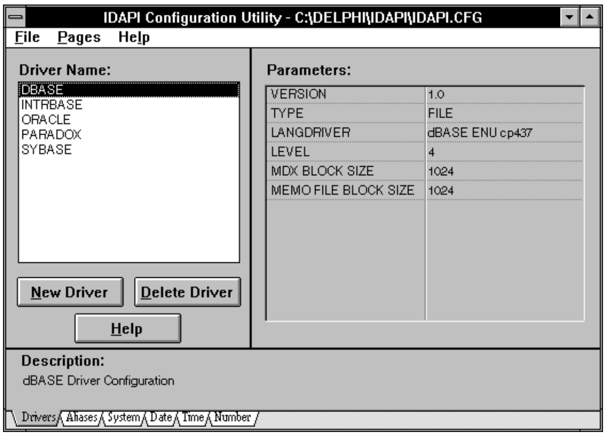
Using the BDE configuration utility

Database Engine Configuration

The Borland Database Engine configuration utility (BDECFG.EXE) enables you to configure BDE aliases and change the settings reflecting your specific environment in the BDE configuration file, IDAPI.CFG.

To run the BDE Configuration Utility, double-click the BDE configuration utility icon in the Delphi program group. The BDE Configuration Utility opens:

Figure B.1 BDE Configuration Utility main window



Driver Name:

```
DBASE
INTRBASE
ORACLE
PARADOX
SYBASE
```

Parameters:

VERSION	1.0
TYPE	FILE
LANGDRIVER	dBASE ENU cp437
LEVEL	4
MDX BLOCK SIZE	1024
MEMO FILE BLOCK SIZE	1024

Description:
dBASE Driver Configuration

Creating and managing aliases

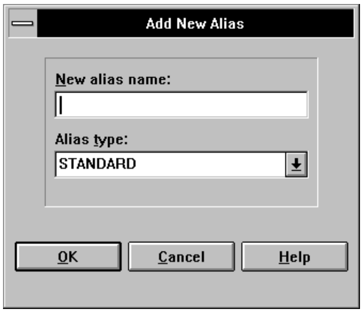
It is usually best to use an alias in your application instead of a hard-coded file or directory name. Setting up a standard alias consists of assigning a name to, and specifying the path name for a directory containing Paradox or dBASE files or the directory path and database name on a SQL server.

Adding a new alias

To add a new alias,

1 Select the Alias Manager (Aliases page) and choose the New Alias button. The Add New Alias dialog box appears. The type can be STANDARD or SQL-specific.

Figure B.2 Sample Add New Alias dialog box

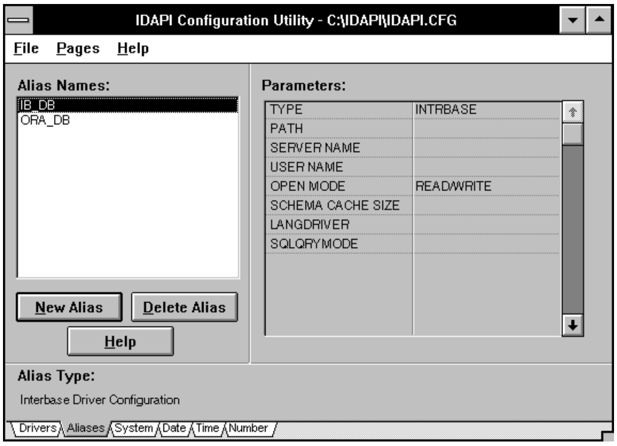


New alias name:

Alias type: STANDARD

2 Enter the new alias name and select the SQL-specific alias type. Then choose OK to begin the setup process. The Alias Manager displays all the configuration parameters you can change to customize the new alias.

Figure B.3 Customizing the new alias



Alias Names:
IB_DB
ORA_DB

Parameters:

TYPE	INTRBASE
PATH	
SERVER NAME	
USER NAME	
OPEN MODE	READ/WRITE
SCHEMA CACHE SIZE	
LANGDRIVER	
SQLQRYMODE	

Alias Type:
Interbase Driver Configuration

3 If desired, edit the settings for the category you selected. If you leave any categories blank, the Alias Manager assumes you want to use the default for driver type.

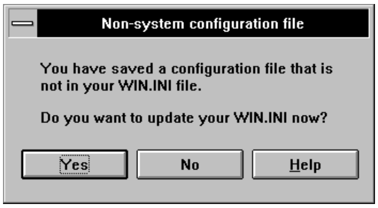
4 When you are finished, select File | Save to save the new alias in the default configuration file; select File | Save as to save the new alias in a configuration file with a different name.

Note The other pages contain settings that can also be customized. See online help for specifics.

Page	Settings modified
Driver Manager	Those BDE uses to determine how an application creates, sorts, and handles tables.
System Manager	Those BDE uses to start an application.
Date Manager	Those used to convert string values into date values.
Time Manager	Those used to convert string values into time values.
Number Manager	Those used to convert string values to number values.

If you save the new alias in a configuration file with a different name, the BDE Configuration Utility displays:

Figure B.4 BDE non-system configuration dialog box


Non-system configuration file

You have saved a configuration file that is not in your WIN.INI file.

Do you want to update your WIN.INI now?

Choose Yes if you want to activate this configuration file next time you start your application. Choose No if you want to keep using the current default configuration file.

Your changes take effect the next time you start your application.

Modifying an existing alias

To modify an existing alias,

1 Scan the list of Alias Names available through the current configuration file. If the alias you want to modify was stored in a different configuration file, use File | Open to load that configuration file.

2 Highlight the name of the alias you want to modify. The configuration for that alias appears in the Parameters section of the Alias Manager page.

3 Highlight the configuration parameter you want to change, and enter the desired value. If you leave any categories blank, the Alias Manager assumes you want to use the driver's default value.

4 When you are finished, select File | Save to save the new alias in the default configuration file; select File | Save As to save the new alias in a configuration file with a different name.

When you modify a driver parameter, all aliases that use the default setting for that parameter inherit the new setting.

Your changes take effect the next time you start your application.

Deleting an alias

To delete an alias,

1 Scan the list of Alias Names available through the current configuration file. If the alias you want to delete was stored in a different configuration file, use File | Open to load that configuration file.

2 Highlight the name of the alias you want to modify, and select the Delete Alias button.

3 Select File | Save to save your changes in the default configuration file; select File | Save As to save your changes in a different configuration file.

Note If an application attempts to use an alias that has been deleted, Delphi will raise an exception. If a Delphi form was saved with a table or query open, Delphi will attempt to open the dataset when the form is loaded and the exception will occur at that time. In many cases, modifying an alias can also cause an exception in Delphi forms if the changes to the alias require changes to the Delphi application.

Appendix

C

Using local SQL

The BDE enables limited access to database tables through local SQL (also called "client-based SQL"). Local SQL is a subset of ANSI-standard SQL enhanced to support Paradox and dBASE naming conventions for tables and fields (called "columns" in SQL). Two categories of SQL statements are supported:

- Data Manipulation Language (DML) for selecting, inserting, updating, and deleting table data.
- Data Definition Language (DDL) for creating, altering, and dropping tables, and for creating and dropping indexes.

This appendix describes naming conventions, syntax enhancements, and syntax limitations for local SQL. For a complete introduction to ANSI-standard SQL, see one of the many third-party books available at your local computer book store.

Naming conventions for tables

ANSI-standard SQL confines each table name to a single word comprised of alphanumeric characters and the underscore symbol (_). Local SQL is enhanced to support full file and path specifications for table names. Table names with path or file-name extensions must be enclosed in single or double quotes. For example,

```
SELECT * FROM 'PARTS.DBF'

SELECT * FROM "C:\SAMPLE\PARTS.DBF"
```

Local SQL also supports BDE aliases for table names. For example,

```
SELECT * FROM :PDOX:TABLE1
```

Finally, local SQL permits table names to duplicate SQL keywords as long as those table names are enclosed in single or double quotes. For example,

```
SELECT PASSID FROM "PASSWORD"
```

Naming conventions for columns

ANSI-standard SQL confines each column name to a single word of alphanumeric characters and the underscore symbol (_). Local SQL is enhanced to support Paradox and dBASE multi-word column names and column names that duplicate SQL keywords as long as those column name are

- Enclosed in single or double quotes.
- Prefaced with an SQL table name or table correlation name.

For example, the following column name is two words:

```
SELECT E."Emp Id" FROM EMPLOYEE E
```

In the next example, the column name duplicates the SQL DATE keyword:

```
SELECT DATELOG."DATE" FROM DATELOG
```

Data manipulation

With some restrictions, local SQL supports the following statements for data manipulation:

SELECT, for retrieving existing data

INSERT, for adding new data to a table

UPDATE, for modifying existing data

DELETE, for removing existing data from a table

The following sections describe parameter substitution, aggregate, string, and date functions, and operators available to DML statements in local SQL.

Parameter substitutions in DML statements

Variables or parameter markers (?) can be used in DML statements in place of values. Variables must always be preceded by a colon (:). For example,

```
SELECT LAST_NAME, FIRST_NAME
FROM "CUSTOMER.DB"
WHERE LAST_NAME > :var1 AND FIRST_NAME < :var2
```

Supported set (aggregate) functions

The following ANSI-standard SQL set (or “aggregate”) functions are available to local SQL for use with data retrieval:

- SUM(), for totaling all numeric values in a column
- AVG(), for averaging all non-NULL numeric values in a column
- MIN(), for determining the minimum value in a column

- MAX(), for determining the maximum value in a column
- COUNT(), for counting the number of values in a column that match specified criteria

Note Expressions are not allowed in set functions.

Supported string functions

Local SQL supports the following ANSI-standard SQL string manipulation functions for retrieval, insertion, and updating:

- UPPER(), to force a string to uppercase
- LOWER(), to force a string to lowercase
- SUBSTRING(), to return a specified portion of a string
- TRIM(), to remove repetitions of a specified character from the left, right, or both sides of a string

Supported date function

Local SQL supports the EXTRACT() function for isolating a single numeric field from a date/time field on retrieval using the following syntax:

```
EXTRACT (extract_field FROM field_name)
```

For example, the following statement extracts the year value from a DATE field:

```
SELECT EXTRACT(YEAR FROM HIRE_DATE)
FROM EMPLOYEE
```

You can also extract MONTH, DAY, HOUR, MINUTE, and SECOND using this function.

Note EXTRACT does not support the TIMEZONE_HOUR or TIMEZONE_MINUTE clauses.

Supported operators

Local SQL supports the following arithmetic operators:

```
+, -, *, /
```

Local SQL supports the following comparison operators:

```
<, >, =, <>, IS NULL
```

Local SQL supports the following logical operators:

```
AND, OR, NOT
```

Local SQL supports the following string concatenation operator:

```
||
```

Using SELECT

The SELECT statement is used to retrieve data from one or more tables. A SELECT that retrieves data from multiple tables is called a "join." Local SQL supports the following form of the SELECT statement:

```
SELECT [DISTINCT] column_list
FROM table_reference
[WHERE search_condition]
[ORDER BY order_list]
[GROUP BY group_list]
[HAVING having_condition]
```

Except as noted below, all clauses are handled as in ANSI-standard SQL. Clauses in square brackets are optional.

The *column_list* indicates the columns from which to retrieve data. For example, the following statement retrieves data from two columns:

```
SELECT PART_NO, PART_NAME
FROM PARTS
```

Using the FROM clause

The FROM clause specifies the table or tables from which to retrieve data. *table_reference* can be a single table, a comma-delimited list of tables, or can be an inner or outer join as specified in the SQL-92 standard. For example, the following statement specifies a single table:

```
SELECT PART_NO
FROM "PARTS.DBF"
```

The next statement specifies a left outer join for *table_reference*:

```
SELECT * FROM PARTS LEFT OUTER JOIN INVENTORY
ON PARTS.PART_NO = INVENTORY.PART_NO
```

Using the WHERE clause

The optional WHERE clause reduces the number of rows returned by a query to those that match the criteria specified in *search_condition*. For example, the following statement retrieves only those rows with PART_NO greater than 543:

```
SELECT * FROM PARTS
WHERE PART_NO > 543
```

The WHERE clause can now include the IN predicate, followed by a parenthesized list of values. For example, the next statement retrieves only those rows where a part number matches an item in the IN predicate list:

```
SELECT * FROM PARTS
WHERE PART_NO IN (543, 544, 546, 547)
```

Important A *search_condition* cannot include subqueries.

Using the ORDER BY clause

The ORDER BY clause specifies the order of retrieved rows. For example, the following query retrieves a list of all parts listed in alphabetical order by part name:

```
SELECT * FROM PARTS
ORDER BY PART_NAME ASC
```

The next query retrieves all part information ordered in descending numeric order by part number:

```
SELECT * FROM PARTS
ORDER BY PART_NO DESC
```

Calculated fields can be ordered by correlation name or ordinal position. For example, the following query orders rows by FULL_NAME, a calculated field:

```
SELECT LAST_NAME || ', ' || FIRST_NAME AS FULL_NAME, PHONE
FROM CUSTOMERS
ORDER BY FULL_NAME
```

Using the GROUP BY clause

The GROUP BY clause specifies how retrieved rows are grouped for aggregate functions. In local SQL, any column names that appear in the GROUP BY clause must also appear in the SELECT clause.

Heterogeneous joins

Local SQL supports joins of tables in different database formats; such a join is called a “heterogeneous join.” For example, it is possible to retrieve data from a Paradox table and a dBASE table as follows:

```
SELECT DISTINCT C.CUST_NO, C.STATE, O.ORDER_NO
FROM "CUSTOMER.DB" C, "ORDER.DBF" O
WHERE C.CUST_NO = O.CUST_NO
```

You can also use BDE aliases in place of table names.

Using INSERT

In local SQL, INSERT is restricted to a list of values:

```
INSERT INTO CUSTOMER (FIRST_NAME, LAST_NAME, PHONE)
VALUES(:fname, :lname, :phone_no)
```

Insertion from one table to another through a subquery is not allowed.

Using UPDATE

There are no restrictions on or extensions to the ANSI-standard UPDATE statement.

Using DELETE

There are no restrictions on or extensions to the ANSI-standard DELETE statement.

Data definition

Local SQL supports data definition language (DDL) for creating, altering, and dropping tables, and for creating and dropping indexes. All other ANSI-standard SQL DDL statements are not supported. In particular, views are not supported.

Local SQL does not permit the substitution of variables for values in DDL statements.

Using CREATE TABLE

CREATE TABLE is supported with the following limitations:

- Column definitions based on domains are not supported.
- Constraints are limited to PRIMARY KEY for Paradox. Constraints are unsupported in dBASE.

For example, the following statement creates a Paradox table with a PRIMARY KEY constraint on the LAST_NAME and FIRST_NAME columns:

```
CREATE TABLE "employee.db"
(
LAST_NAME CHAR(20),
FIRST_NAME CHAR(15),
SALARY NUMERIC(10,2)
DEPT_NO SMALLINT,
PRIMARY KEY(LAST_NAME, FIRST_NAME)
)
```

The same statement for a dBASE table should omit the PRIMARY KEY definition:

```
CREATE TABLE "employee.db"
(
LAST_NAME CHAR(20),
FIRST_NAME CHAR(15),
SALARY NUMERIC(10,2)
DEPT_NO SMALLINT
)
```

The following table lists SQL syntax for data types used with CREATE TABLE, and describes how those types are mapped to Paradox and dBASE types by the BDE:

Table C.1 Data type mappings

SQL Syntax	**BDE Logical**	**Paradox**	**dBASE**
SMALLINT	fldINT16	fldPDXSHORT	fldDBNUM
INTEGER	fldINT32	fldPDXLONG	fldDBNUM
DECIMAL(x,y)	fldBCD	fldPDXBCD	N/A
NUMERIC(x,y)	fldFLOAT	fldPDXNUM	fldDBNUM(x,y)
FLOAT(x,y)	fldFLOAT	fldPDXNUM	fldDBFLOAT(x,y)
CHARACTER(n)	fldZSTRING	fldPDXALPHA	fldDBCHAR
x = precision (default: specific to driver); y = scale (default: 0); n = length in bytes (default: 0); s = BLOB subtype (default: 1)			

Table C.1 Data type mappings

SQL Syntax	BDE Logical	Paradox	dBASE
VARCHAR(n)	fldZSTRING	fldPDXALPHA	fldDBCHAR
DATE	fldDATE	fldPDXDATE	fldDBDATE
BOOLEAN	fldBOOL	fldPDXBOOL	fldDBBOOL
BLOB(n,s)	See Subtypes below	See Subtypes below	See subtypes below
TIME	fldTIME	fldPDXTIME	N/A
TIMESTAMP	fldTIMESTAMP	fldPDXTIMESTAMP	N/A
MONEY	fldFLOAT, fldstMONEY	fldPDXMONEY	fldDBFLOAT(20,4)
AUTOINC	fldINT32, fldstAUTOINC	fldPDXAUTOINC	N/A
BYTES(n)	fldBYTES(n)	fldPDXBYTES	fldDBBYTES (in-memory tables only)
x = precision (default: specific to driver); y = scale (default: 0); n = length in bytes (default: 0); s = BLOB subtype (default: 1)			

The following table specifies how BLOB subtypes translate from SQL to Paradox and dBASE through the BDE:

Table C.2 BLOB subtype mappings

SQL Subtype	BDE Logical	Paradox	dBASE
1	fldstMEMO	fldPDXMEMO	fldDBMEMO
2	fldstBINARY	fldPDXBINARY	fldDBBINARY
3	fldstFMTMEMO	fldPDXFMTMEMO	N/A
4	fldstOLEOBJ	fldPDXOLEBLOB	fldDBOLEBLOB
5	fldstGRAPHIC	fldPDXGRAPHIC	N/A

Using ALTER TABLE

Local SQL supports the following subset of the ANSI-standard ALTER TABLE statement. You can add new columns to an existing table using this ALTER TABLE syntax:

```
ALTER TABLE table ADD column_name data_type [, ADD column_name data_type . . .]
```

For example, the following statement adds a column to a dBASE table:

```
ALTER TABLE "employee.dbf" ADD BUILDING_NO SMALLINT
```

You can delete existing columns from a table using the following ALTER TABLE syntax:

```
ALTER TABLE table DROP column_name [, DROP column_name . . .]
```

For example, the next statement drops two columns from a Paradox table:

```
ALTER TABLE "employee.db" DROP LAST_NAME, DROP FIRST_NAME
```

ADD and DROP operations can be combined in a single statement. For example, the following statement drops two columns and adds one:

```
ALTER TABLE "employee.dbf" DROP LAST_NAME, DROP FIRST_NAME, ADD FULL_NAME CHAR[30]
```

Using DROP TABLE

DROP TABLE deletes a Paradox or dBASE table. For example, the following statement drops a Paradox table:

```
DROP TABLE "employee.db"
```

Using CREATE INDEX

CREATE INDEX enables users to create indexes on tables using the following syntax:

```
CREATE INDEX index_name ON table_name (column [, column . . .])
```

Using CREATE INDEX is the only way to create indexes for dBASE tables. For example, the following statement creates an index on a dBASE table:

```
CREATE INDEX NAMEX ON "employee.dbf" (LAST_NAME)
```

Paradox users can only create secondary indexes with CREATE INDEX. Primary Paradox indexes can only be created by specifying a PRIMARY KEY constraint when creating a new table with CREATE TABLE.

Using DROP INDEX

Local SQL provides the following variation of the ANSI-standard DROP INDEX statement for deleting an index. It is modified to support dBASE and Paradox file names.

```
DROP INDEX table_name.index_name | PRIMARY
```

The PRIMARY keyword is used to delete a primary Paradox index. For example, the following statement drops the primary index on EMPLOYEE.DB:

```
DROP INDEX "employee.db".PRIMARY
```

To drop any dBASE index, or to drop secondary Paradox indexes, provide the index name. For example, the next statement drops a secondary index on a Paradox table:

```
DROP INDEX "employee.db".NAMEX
```

Appendix

D

The MAST database

This appendix describes the MAST sample database provided with Delphi and used for the MASTAPP sample application. The basic MAST database is in Paradox format. A SQL version that uses the Local InterBase Server is also provided. See README.TXT for a complete list of sample databases and applications.

MAST is the fictional Marine Adventures and Sunken Treasures company. MAST's customers are dive shops around the world. They sell products and supplies to these shops; the shops place orders for equipment. The following tables are used to track MAST's sales:

Table D.1 MAST tables

Table name	Information in table
CUSTOMER.DB	Customer dive shop data, including customer number, name, etc.
EMPLOYEE.DB	MAST employee information
ITEMS.DB	Specific items that makes up customer orders
NEXTORD.DB	Table that maintains the next unique order number
ORDERS.DB	Orders placed by customer dive shops
PARTS.DB	Inventory information about items on hand at MAST
VENDOR.DB	MAST suppliers that sell goods to MAST

Each table contains a primary key. To link tables, some fields and data must be duplicated among tables. A table must have a primary key or secondary index assigned to the duplicate field before it can be linked to another table. Fields that are duplicated between tables use referential integrity to make sure their values match in all tables.

The following tables describe each of the MAST database tables by showing the fields contained in the table, the type of each field, the size of the alphanumeric fields, and the fields that are key fields.

The CUSTOMER table is structured as follows:

Field	Type	Size	Key
CustNo	Numeric		*
Company	Alphanumeric	30	
Addr1	Alphanumeric	30	
Addr2	Alphanumeric	30	
City	Alphanumeric	15	
State	Alphanumeric	20	
Zip	Alphanumeric	10	
Country	Alphanumeric	20	
Phone	Alphanumeric	15	
FAX	Alphanumeric	15	
TaxRate	Numeric		
Contact	Alphanumeric	20	
LastInvoiceDate	Timestamp		

In CUSTOMER, the CustNo field is a primary key because orders in the ORDERS table must be linked to customers. The secondary index on Company is named "ByCompany". ORDERS itself contains only information about each order placed by a customer.

The structure of the EMPLOYEE table is as follows:

Field	Type	Size	Key
EmpNo	Long integer		*
LastName	Alphanumeric	20	
FirstName	Alphanumeric	15	
PhoneExt	Alphanumeric	4	
HireDate	Timestamp		
Salary	Numeric		

The ITEMS table is structured as follows:

Field	Type	Size	Key
ItemNo	Numeric		*
OrderNo	Numeric		
PartNo	Numeric		
Qty	Long integer		
Discount	Numeric		

OrderNo is a secondary index and is used in the master-detail link in the Orders form.

The NEXTORD tables is a single-column table used to generate unique, sequential order numbers. In a multi-user environment using Paradox tables, the only way to guarantee that an order number is unique is to store the last-used number in a table, and increment it each time a new number is fetched. The table is structured as follows:

Field	Type	Size	Key
NewKey	Numeric		

The ORDERS table is structured as follows:

Field	Type	Size	Key
OrderNo	Numeric		*
CustNo	Numeric		
SaleDate	Timestamp		
ShipDate	Timestamp		
EmpNo	Long integer		
ShipToContact	Alphanumeric	20	
ShipToAddr1	Alphanumeric	30	
ShipToAddr2	Alphanumeric	30	
ShipToCity	Alphanumeric	15	
ShipToState	Alphanumeric	20	
ShipToZip	Alphanumeric	10	
ShipToCountry	Alphanumeric	20	
ShipToPhone	Alphanumeric	15	
ShipVIA	Alphanumeric	7	
PO	Alphanumeric	15	
Terms	Alphanumeric	6	
PaymentMethod	Alphanumeric	7	
ItemsTotal	Money		
TaxRate	Numeric		
Freight	Money		
AmountPaid	Money		

In ORDERS, the link to CUSTOMER is through the CustNo field. A secondary index is defined on CustNo to ensure that ORDERS and CUSTOMERS can be sorted and linked in the same order. To be sure that values entered as CustNo in ORDERS match exactly one record in CUSTOMER, referential integrity is used to constrain values entered in the CustNo field to valid customer numbers already in the CUSTOMER table.

The primary key in ITEMS is a composite of the OrderNo and ItemNo fields; these fields also have secondary indexes. This enables ITEMS to link to ORDERS (using OrderNo) in the MASTAPP Orders form. Referential integrity is used to validate the information in PARTS.

The PARTS table tracks the inventory of products. PARTS is structured as follows so that it can be linked to VENDORS:

Field	Type	Size	Key
PartNo	Numeric		*
VendorNo	Numeric		
Description	Alphanumeric	30	
OnHand	Numeric	20	
OnOrder	Numeric		
Cost	Money		
ListPrice	Money		

PartNo is the primary key for this table. Because PartNo is a secondary index, the two tables can be linked. VendorNo is the primary key of VENDORS, so it is a secondary index in PARTS and is defined to use referential integrity to make sure values entered in VendorNo match a single VendorNo entry in the VENDORS table.

The VENDORS table has the following structure:

Field	Type	Size	Key
VendorNo	Numeric		*
VendorName	Alphanumeric	30	
Address1	Alphanumeric	30	
Address2	Alphanumeric	30	
City	Alphanumeric	20	
State	Alphanumeric	20	
Zip	Alphanumeric	10	
Country	Alphanumeric	15	
Phone	Alphanumeric	15	
FAX	Alphanumeric	15	
Preferred	Logical		

Using these tables, a customer's information is entered only once, and then referred to in other tables. Likewise, order, item, inventory, and vendor information is entered only once, reducing storage requirements, and the chance for data entry error.

Index

Database Application Developer's Guide

Delphi™

Borland International, Inc., 100 Borland Way
P.O. Box 660001, Scotts Valley, CA 95067-0001

Printed in the U.S.A.

1E0R195
9596979899-987654321
W1

Contents

Tables

Figures